推酷

编程狂人

Programming Madman

NO.44

关于推酷

推酷是专注于IT圈的个性化阅读社区。我们利用智能算法, 从海量文章资讯中挖掘出高质量的内容,并通过分析用户 的阅读偏好,准实时推荐给你最感兴趣的内容。我们推荐 的内容包含科技、创业、设计、技术、营销等内容,满足 你日常的专业阅读需要。我们针对IT人还做了个活动频道, 它聚合了IT圈最新最全的线上线下活动,使IT人能更方便地 找到感兴趣的活动信息。

关于周刊

《编程狂人》是献给广大程序员们的技术周刊。我们利用技术挖掘出那些高质量的文章,并通过人工加以筛选出来。每期的周刊一般会在周二的某个时间点发布,敬请关注阅读。

本期为精简版 周刊完整版链接:http://www.tuicool.com/mags/5428babcd91b14788b01f242

欢迎下载推酷客户端体验更多阅读乐趣

目录

CSS命名神马的真心难

译者：囧克斯

找到的这篇文章算是对我之前写的 《标签？ID？还是CLASS？》 的再深入。我当时写那篇文章的时候，就有朋友提出了“非语义化”的 class 命名的问题，我当时确实觉得很纠结，简单的想法是“框架性质的表象 class 我没异议……框架的实质是通过降低灵活性达成更广泛的共识，我们个人不要再创造这样的样式就好了”，但没有想到特别好的“套路”，更多的是在实际情况中再 分辨。看过这篇文章，我似乎找到了更好的答案。同时顺着文中提到的 Nicolas 那篇文章看下去，也对 OOCSS、BEM 之类的提法有了更多的认同感。特译给大家参考。

这并不是一篇有关 CSS 架构的文章，也并不是一篇有关命名规范的文章，而关乎我们如何定位元素，关乎命名本身，关乎我们如何把元素及其相关的一段样式连接起来。

10 个开发者里有 9 个都同意：在撰写 CSS 中命名什么的部分是最难的了。因为我们无法预知未来。一个 class 名可以在今天完美的应景，但是明天设计发生改变了，可能就不适用了。所以我们需要提炼应景的标记和样式。嗷~

如何面对这一状况呢？那便是让命名尽量显得不太会改的样子。

我们通常会根据三类情况给定一个 class 名：

- 功能性 class 名
- 内容性 class 名
- 展示性 class 名

这几类 class 名是趋向于稳定特质的。如果我们遵循这些命名原则，就会显得更明智，而且我们的 CSS 会更好的适应未来的改变。

功能性 class 名

```
<button class="positive-button">Send Message</button>
```

功能性 class 名例如 positive-button、important-text 或 selected-tab。这些元素的样式是基于其功能或含义的。所以其 class 名、样式及这样引用样式的理由，都是强连接的。因此 class 名和样式是相关的。

因为有这些强连接，所以样式是几乎不会被改变的。如果你真的要改变一个 positive-button 的样子，那这个改变也是每个肯定语气的按钮都要改变的。如果你的设计师的想法是改变“肯定语气的按钮”，而不是设置菜单里“增加用户”的按钮，那么这件事就很轻松且易于维护。你考虑的不是哪个独立的页面，而是整个系统。

功能性的 class 名很棒。只要有这个可能，这应该就是你想要撰写样式的方式。但是功能性的 class 名不是所有情况都适用的。

设计并不一定都有逻辑性。当我们讨论按钮的时候，给出一个功能性的 class 名是很容易的。大部分情况下其功能和样式也紧密相关。但是我们撰写的其它 99% 的样式都不太容易给出这样的例子。有的时候这个块区域需要一个内阴影，因为这样看起来很漂亮；有的时候图标需要在 hover 的时候长大一点因为这样很 cute；有的时候文本需要是橙色的，好吧，因为它就是橙色的。一个网站不是每个视觉的部分背后都有功能性的理由，这无可厚非。

所以开发者该怎么应对呢？我们站在了这个十字路口。如果我们不能想出一个合适的功能性 class 名，那么我们不妨基于内容、或者展现给它起个名字。它们也各自暗示着不同的可维护性。

内容性 class 名

```
<button class="submit-button">Send Message</button>
```

基于内容的 class 名是描述它们包含的内容的 class 名。如果你曾经见到过类似 submit-button、intro-text 或 profile-photo 的 class 名，那些名字就是基于内容起的。

这些 class 名感觉很干净。它们让你的内容 (HTML) 和样式 (CSS) 之间保持简洁的分离。理论上，这可以让你完全改版网站的样式和感官而无需触碰到 HTML。CSS Zen Garden 就是这样的。

现在我们回到最初的问题：“这样做的好处是什么？”

我从来没有被要求改版一个网站而不触碰 HTML。变化是存在各种可能的。当然一些 HTML 可以作为后台系统的集成成果固定下来，但是随着你对 HTML 的失控，这时你还是需要写一些非理想化的 hacky CSS 来应付设计的变化。想想看，CSS Zen Garden 更多的是一个 CSS 技术演示而不是一个可维护的 CSS 的例子。没有必要一味追求在改版的时候只改 CSS。

当你开发一个小网站的时候，内容性的 class 名非常好用。而随着你的网站不断成长，它就感觉越来越不合适了。它们并不易于样式重用。如果你的 login-button 和 submit-button 看起来一样该怎么处理呢？在你的 CSS 架构里该如何展示这些东西？为保持展现样式块，你不得不写一堆用逗号分隔开的选择器，或者通过预处理器展开。这些组织方式对于大型的项目来说都比较困难。

除非有更好的方式重用样式块……

展示性 class 名

```
<button class="green-button">Send Message</button>
```

展示性 class 名用诸如 green-button、big-text 或 squiggle-border 的方式描述一个元素。其名字本身就是对样式的描述。

这些 class 是有助于代码复用的。它们不关心是否用在产品标题上还是名户名或页头。它们只知道这会让文字变大加粗。同时这样的方式还有一个好处是可以优雅的扩展。如你 开发一个新组件的时候，你可以把现成的样式贴在你的新标签上。你无须担心在已有的架构中产生并适配新的样式，因为你使用的都是已有的样式。

展示性 class 名也非常易于自我描述。一个开发者在审查代码的时候，round-image 会比 profile-photo 更多的推断出这个元素的样子。

会有争议认为展示性 class 增加了维护成本。因为它模糊了标记和展现之间的界限，很多设计的改变都将会导致 HTML 的改变。如果你预见到了这方面的问题，那么请谨慎的使用。

……但这不是语义化的！

展示性 class 的名声并不好。尽管很多人回避它们因为它们“不是语义化的”，但这里是存在误区的，也被 Nicolas Gallagher 质疑。重要的是区分“语义化的 HTML”和“语义化的 class”。Nicolas 说的非常好：

撰写“语义化的 HTML”的原则是现代化、专业化的前端开发的基础。大部分的语义化都关乎现有的或预期的内容的本质…… ……不过并不是所有的语义化都需要源自内容的。Class 名可以是“非语义化的”。不管使用什么名字，它们都有意义，都有目的。Class 名的语义化是不同于那些 HTML 元素的。

如我写的这些，“非语义化”这个词下面是有红色波浪线的。非语义化的 class 名并不是问题。每个 class 名都有背后的意义。在你写 class 名时，不必刻意追求它是最“语义上适合的” class 名，而要创建为开发者和未来的你提供尽量多信息的 class 名。

总结

功能性 class 名通常是你的最佳选择。当你能够使用它们的时候就尽量使用。如果你无法提取出完全功能性的名字，可以考虑你的项目的本质及其发展。原则上，内容性 class 名更适合小型站点；而展示性 class 名更适合大型站点。

开发者会很在意这种用法。没有人希望一个项目变得难以维护，但是每个人都有不同的 想法通过 class 名来应对这些特殊情况。这时不妨思考一下我们使用的不同类型 class 名的本质，问问自己这样做是否更好的帮助你的项目达成目标。

原译文链接：http://jiongks.name/blog/naming-css-stuff-is-really-hard/

原文链接：http://seesparkbox.com/foundry/naming_css_stuff_is_really_hard

单页应用SEO浅谈

作者：colin

前言

单页应用（Single Page Application）越来越受web开发者欢迎，单页应用的体验可以模拟原生应用，一次开发，多端兼容。单页应用并不是一个全新发明的技术，而是随着互联网的发展，满足用户体验的一种综合技术。

SEO

一直以来，搜索引擎优化（SEO）是开发者容易忽略的部分。SEO是针对搜索（Google、百度、雅虎搜索等）在技术细节上的优化，例如语义、搜索关键词与内容相关性、收录量、搜索排名等。SEO也是同行、市场竞争常用的的营销手段。Google、百度的搜索结果是重要的用户入口，腾讯云（www.qcloud.com）有30%左右的流量来自搜索引擎。因此SEO在品牌、营销、用户量的纬度是非常重要的基础能力。

那么单页应用与传统直出页面在SEO方面有哪些不同之处呢?

单页应用的优点

1. 更好的用户体验，让用户在web感受natvie的速度和流畅；
2. 经典MVC开发模式，前后端各负其责。
3. 一套Server API，多端使用（web、移动APP等）
4. 重前端，业务逻辑全部在本地操作，数据都需要通过A-JAX同步、提交；

对搜索引擎不友好

单页应用实际是把视图（View）渲染从Server交给浏览器，Server只提供JSON格式数据，视图和内容都是通过本地JavaScript来组织和渲染。而搜索搜索引擎抓取的内容，需要有完整的HTML和内容，单页应用架构的站点，并不能很好的支持搜索。

如果站点在用户体验和搜索友好权衡时，如果我们做到更好的体验，也做到友好的搜索支持，既是一箭双雕。

URL中的哈希（#号）

单页应用只有一个页面，视图的变化通常是通过路由（route）来驱动，首先，我们先来谈一谈单页应用的URL中的#号，很多采用单元结构网站的URL都出现了这个符号。

#号在浏览器的URL中是一个锚点，在当前页改变#号的参数，页面会跳转到锚点所在的位置，通过JavaScript我们可以获取到#号后的参数：

location.hash // 获取URL hash

location.hash = "#list" //改变URL hash

改变#号后的参数，页面并不会重载，于是大多数的单页架构网站，都在URL中采用#号来作为当前视图的URL地址，例如：

example.com/#index //首页视图

example.com/#list //列表页视图

example.com/#list/1 //id为1的列表信息的视图

Backbone.js就是通过改变#号参数来组织视图，这里有一个demo可以很直观的体验URL的变化。

看过这个demo，你或许会发现很熟悉的符号#!，Twitter曾在URL使用这个标识。这个标识是Google提出（AJAX 抓取：网站站长和开发人员指南 1）：

因为复杂的单页架构页面，对Google来说抓取比较困难，于是给开发者制定一个规范：

1. 网站提交sitemap给Google；
2. Google发现URL里有#!符号，例如example.com/#!/detail/1，于是Google开始抓取example.com/?_escaped_fragment_=/detail/1；

_escaped_fragment_这个参数是Google指定的命名，如果开发者希望把网站内容提交给Google，就必须通过这个参数生成静态页面。

根据上面的demo，我简单示例一下Google要抓取的页面的样子:

http://119.28.4.22/?escapedfragment_=/detail/1

如此以来，就需要Server通过生成静态的内容以便Google抓取。

以下将简单介绍，单页架构，爬虫访问根目录时如果配置Server端的路由。

判断爬虫

当Google访问119.28.4.22/#!/detail/1 时，会自动转化成http://119.28.4.22/?_escaped_fragment_=/detail/1，以Nginx为例：

```
if ($args ~ _escaped_fragment_) {
   rewrite ^ /api;
}
```

/api为后台服务的接口，已nodejs为例，代理设置如下：

```
upstream nodejs {
    server 127.0.0.1:3000;
}

location /api {
    proxy_set_header X-Request-URI   $request_uri;
    proxy_set_header X-Real-IP       $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host            $host;
    proxy_set_header Port            $server_port;
    proxy_pass http://nodejs;
    proxy_redirect off;
    }
```

如此，我们便将Google的访问重写到/api这个接口，然后在Server的/api处理请求把静态内容输出即可。

sitemap

Gogole的这个规范，必须有sitemap支持，因为有可能单页架构的站点，索引页面也是JavaScript渲染的。提交sitemap时，不用关注_escaped_fragment_这个参数名，只提交带哈希符号的URL即可，例如：

```
<loc>http://119.28.4.22/#!/detail/1</loc>
<changefreq>weekly</changefreq>
<priority>0.5</priority>
</url>
```

结语

技术潮流的步伐很快，单页应用，URL哈希处理也没渲染的方式实际上已经流行了很久，在国外很多用户数据较好的情况下，开发者会选择HTML5 History API的pushstate特性开发，在URL中抛弃#!。但是IE6、7等低端浏览器用户情况较多的网站，#能够很好的兼容。关于采用HTML5 History API来架构单页应用的方案，也欢迎讨论。

原文链接：http://isux.tencent.com/seo-for-single-page-applications.html

王帅：深入PHP内核（二）——SAPI探究

作者：王帅

SAPI是Server Application Programming Interface（服务器应用编程接口）的缩写。PHP通过SAPI提供了一组接口，供应用和PHP内核之间进行数据交互。

简单的讲，就像函数的输入和输出一样，我们通过Linux命令行执行一段PHP代码，本质是Linux的Shell通过PHP的SAPI传入一组参 数，Zend引擎执行后，返回给shell，由shell显示出来的过程。同样的，通过Apache调用PHP，通过Web服务器给SAPI传入数 据，Zend引擎执行后，返回给A-pache，由Apache显示在页面上。

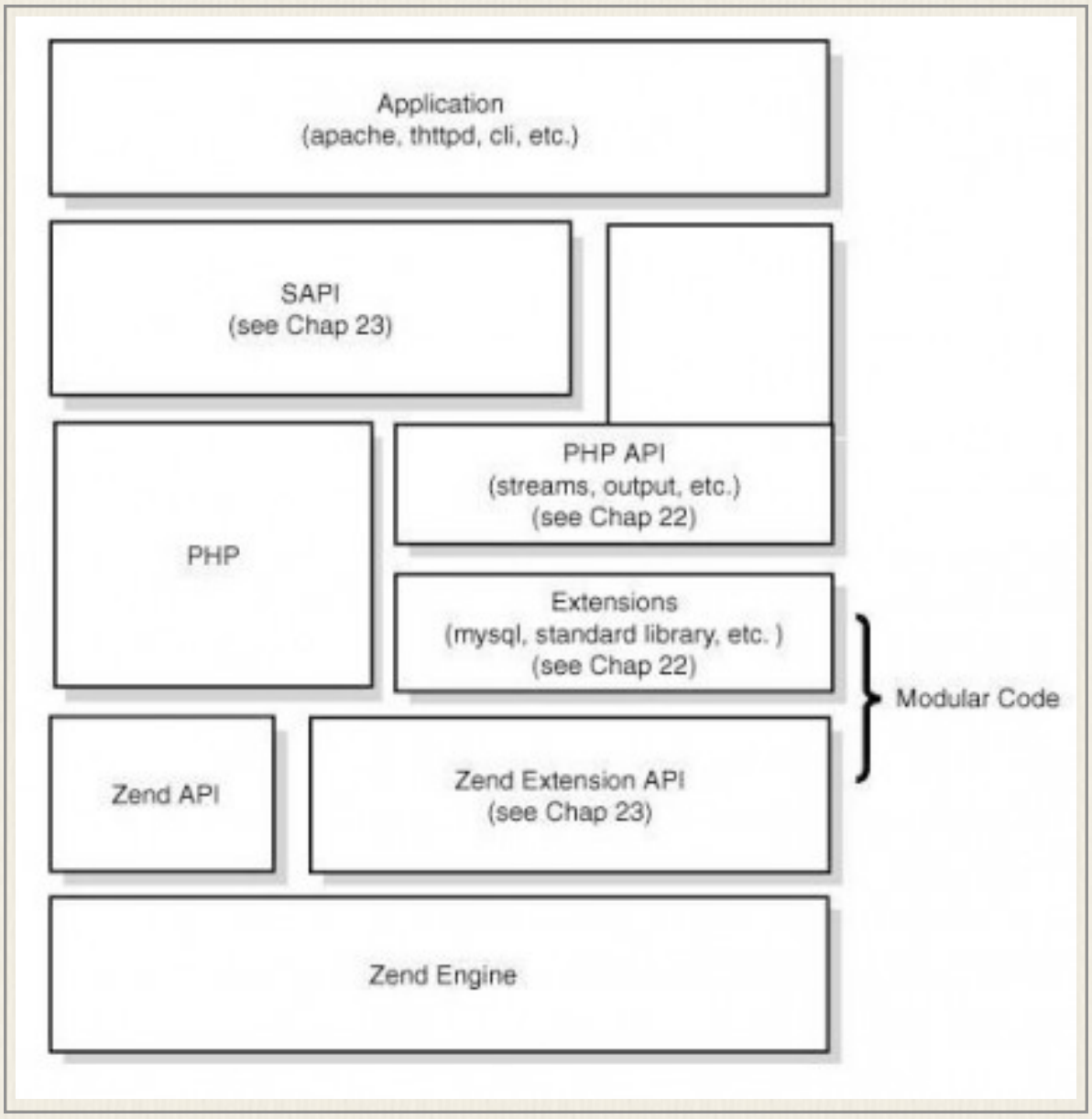


图1. PHP架构图

PHP提供很多种形式的接口，包括apache、apache2filter、apache2handler、caudium、cgi 、cgi-fcgi、cli、cli-server、continuity、embed、isapi、litespeed、milter、nsapi、 phttpd pi3web、roxen、thttpd、tux和webjames。但是常用的只有5种形式，CLI/CGI（命令行）、Multiprocess（多 进程）、Multithreaded（多线程）、FastCGI和Embedded（内嵌）。

PHP提供了一个函数查看当前SAPI接口类型：

string php_sapi_name (void)

PHP的运行和加载

无论使用哪种SAPI，在PHP执行脚本前后，都包含一系列事件：Module的Init(MINT)和Shutdown(MSHUTDOWN)，Request 的Init(RINT)和Shutdown(RSHUTDOWN)。 第一阶段是PHP模块初始化阶段（MINT），可以初始化扩展内部变量、分配资源和注册资源处理器，在整个PHP实例生命周期内，该过程只执行一次。

什么是PHP模块？通过上面的PHP架构图，在PHP中可以使用get_loaded_extensions 函数来查看所有编译并加载的模块/扩展，相当于CLI模式下的php -m。

以PHP的Memcached扩展源代码为例：

```
1.  PHP_MINIT_FUNCTION(memcached) {
2.      zend_class_entry ce;
3.  memcpy(&memcached_object_handlers,zend_get_std_object_handlers(), sizeof(zend_object_handlers));
```

```
4.  memcached_object_handlers.clone_obj = NULL;    /* 执行了一些类似的初始化操作 */
5.  return SUCCESS;
6.  }
```

第二阶段是请求初始化阶段（RINT），在模块初始化并激活后，会创建PHP运行环境，同时调用所有模块注册的RINT函数，调用每个扩展的请求初始化函数，设定特定的环境变量、分配资源或执行其他任务，如审核。

```
1.  PHP_RINIT_FUNCTION(memcached) {
2.      /* 执行一些关于请求的初始化 */
3.      return SUCCESS;
4.  }
```

第三阶段，请求处理完成后，会调用PHP_RSHUTDOWN_FUNCTION进行回收，这是每个扩展的请求关闭函数，执行最后的清理工 作。Zend引擎执行清理过程、垃圾收集、对之前的请求期间用到的每个变量执行unset。请求完成可能是执行到脚本完成，也可能是调用die()或 exit()函数完成

第四阶段，当PHP生命周期结束时候，PHP_MSHUTDOWN_FUNCTION 对模块进行回收处理，这是每个扩展的模块关闭函数，用于关闭自己的内核子系统。

```
1.  PHP_MSHUTDOWN_FUNCTION(memcached) { /* 执行关于模块的销毁工作 */ UNREGISTER_INI_ENTRIES(); return SUCCESS; }
```

常见的运行模式

常见的SAPI模式有五种：

- CLI和CGI模式（单进程模式）
- 多进程模式
- 多线程模式
- FastCGI模式
- 嵌入式

1. CLI/CGI模式

CLI和CGI都属于单进程模式，PHP的生命周期在一次请求中完成。也就是说每次执行PHP脚本，都会执行第二部分讲的四个INT和Shutdown事件。

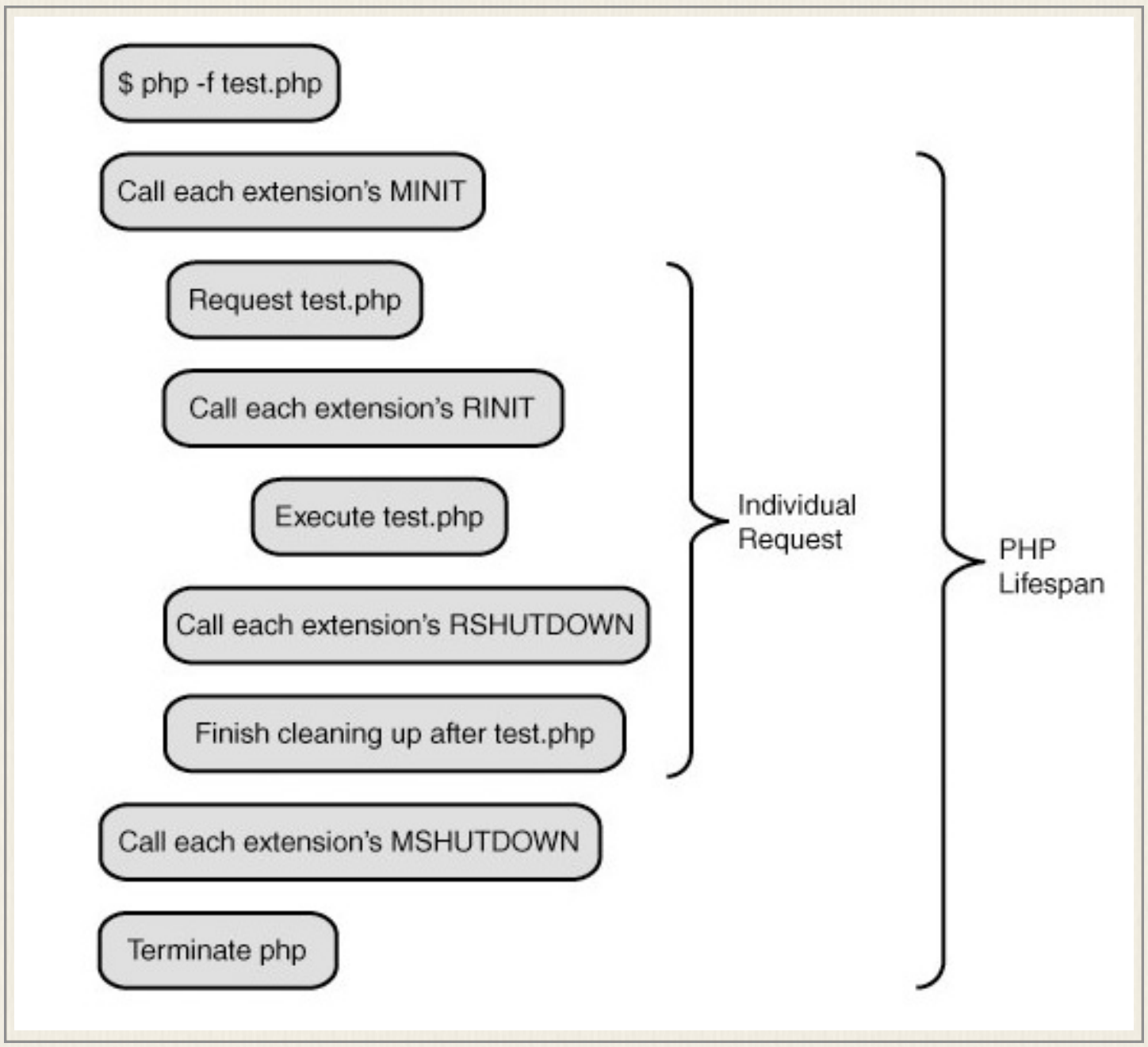


图2. CGI/CLI生命周期

2. 多进程模式（Multiprocess）

多进程模式可以将PHP内置到Web Server中，PHP可以编译成Apache下的prefork MPM模式和APXS模块，当Apache启动后，会fork很多子进程，每个子进程拥有自己独立的进程地址空间。

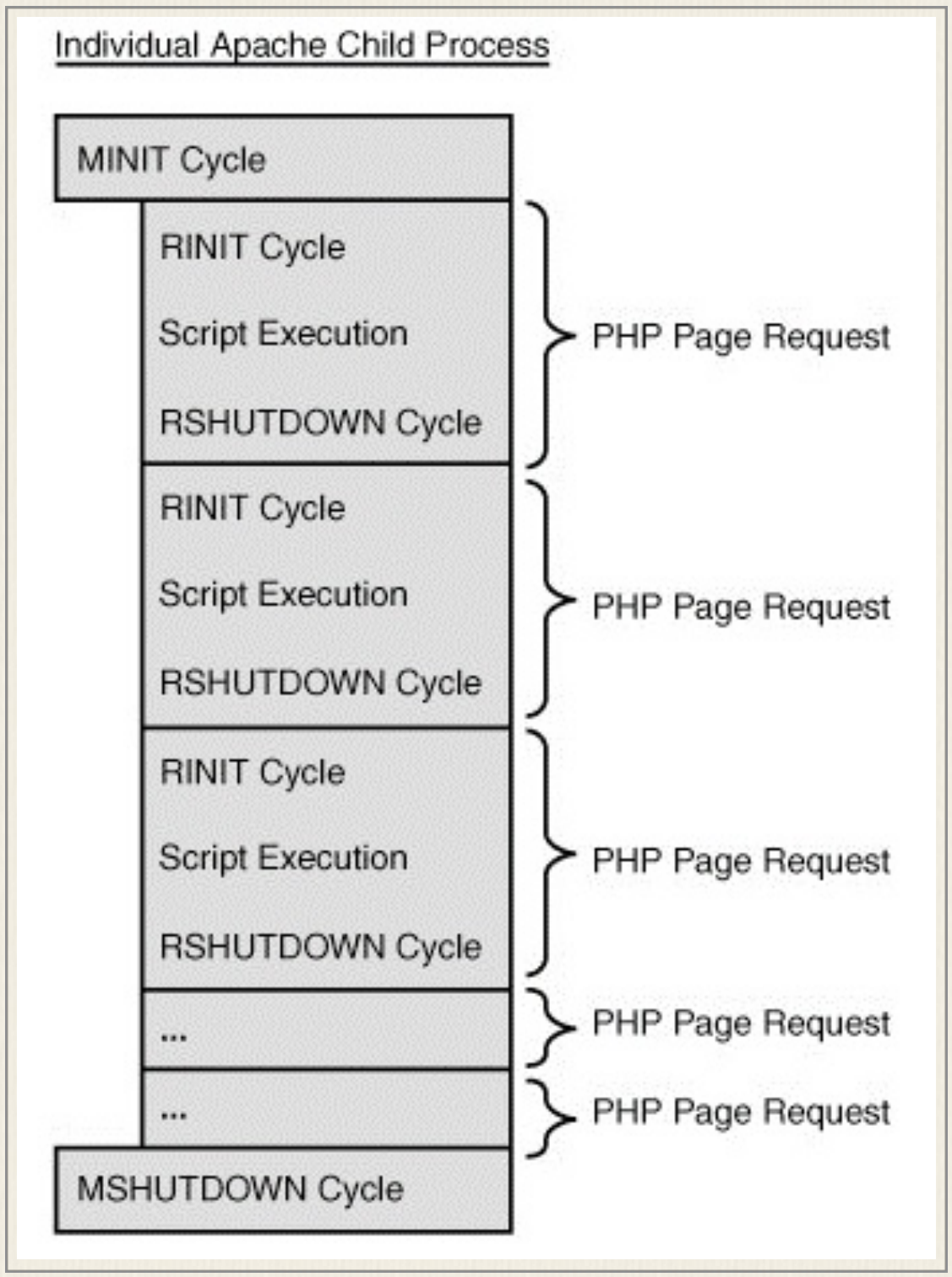


图3. 多进程模式生命周期

在一个子进程中，PHP的生命周期是调用MINT启动后，执行多次请求（RINT/RSHUTDOWN)，在Apache关闭或进程结束后，才会调用MSHUTDOWN进行回收阶段。

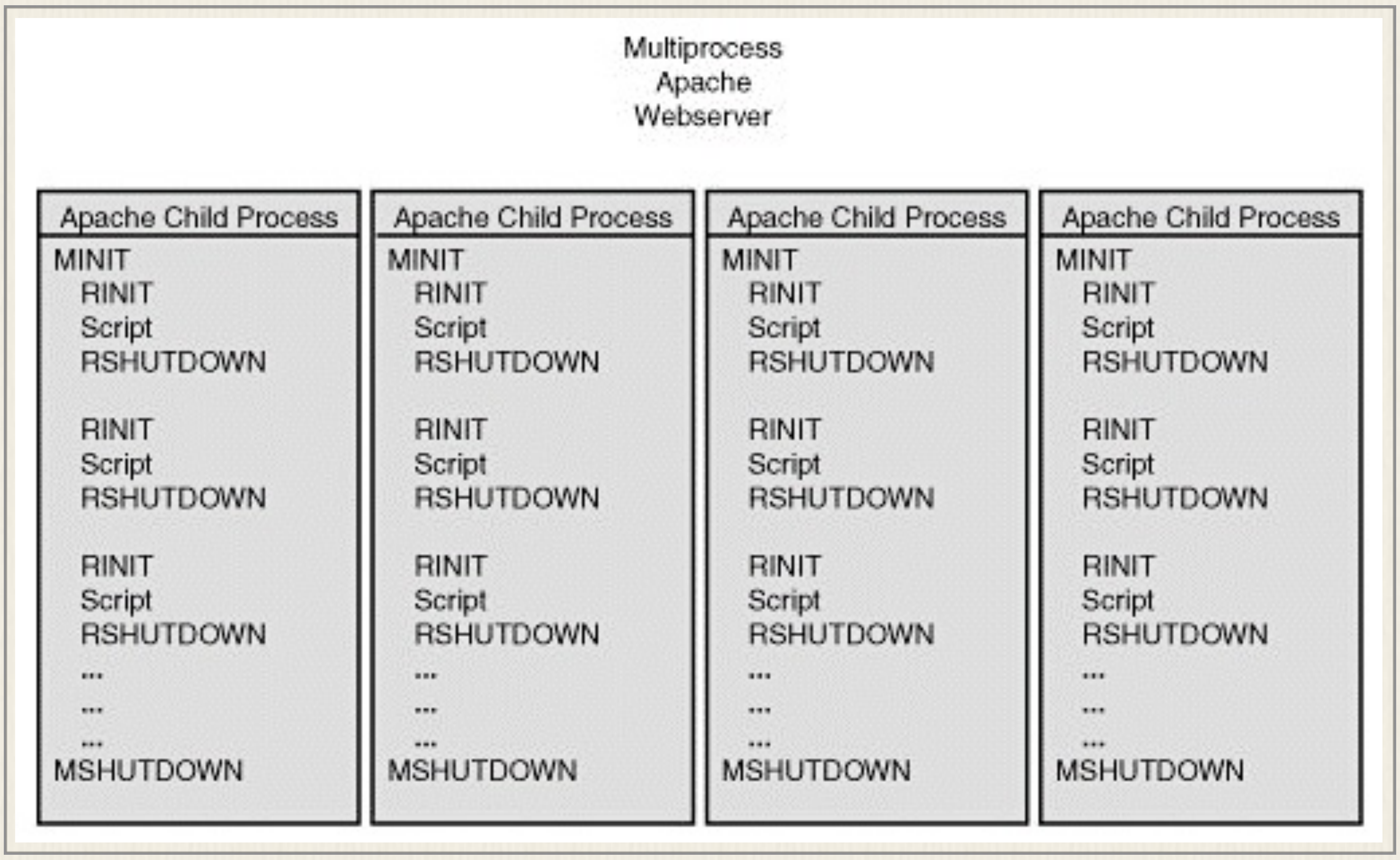


图4. 多进程的生命周期

多进程模型中，每个子进程都是独立运行，没有代码和数据共享，因此一个子进程终止退出和重新生成，不会影响其他子进程的稳定。

3. 多线程模式（Multithreaded)

Apache2的Worker MPM采用了多线程模型，在一个进程下创建多个线程，在同一个进程地址空间执行。

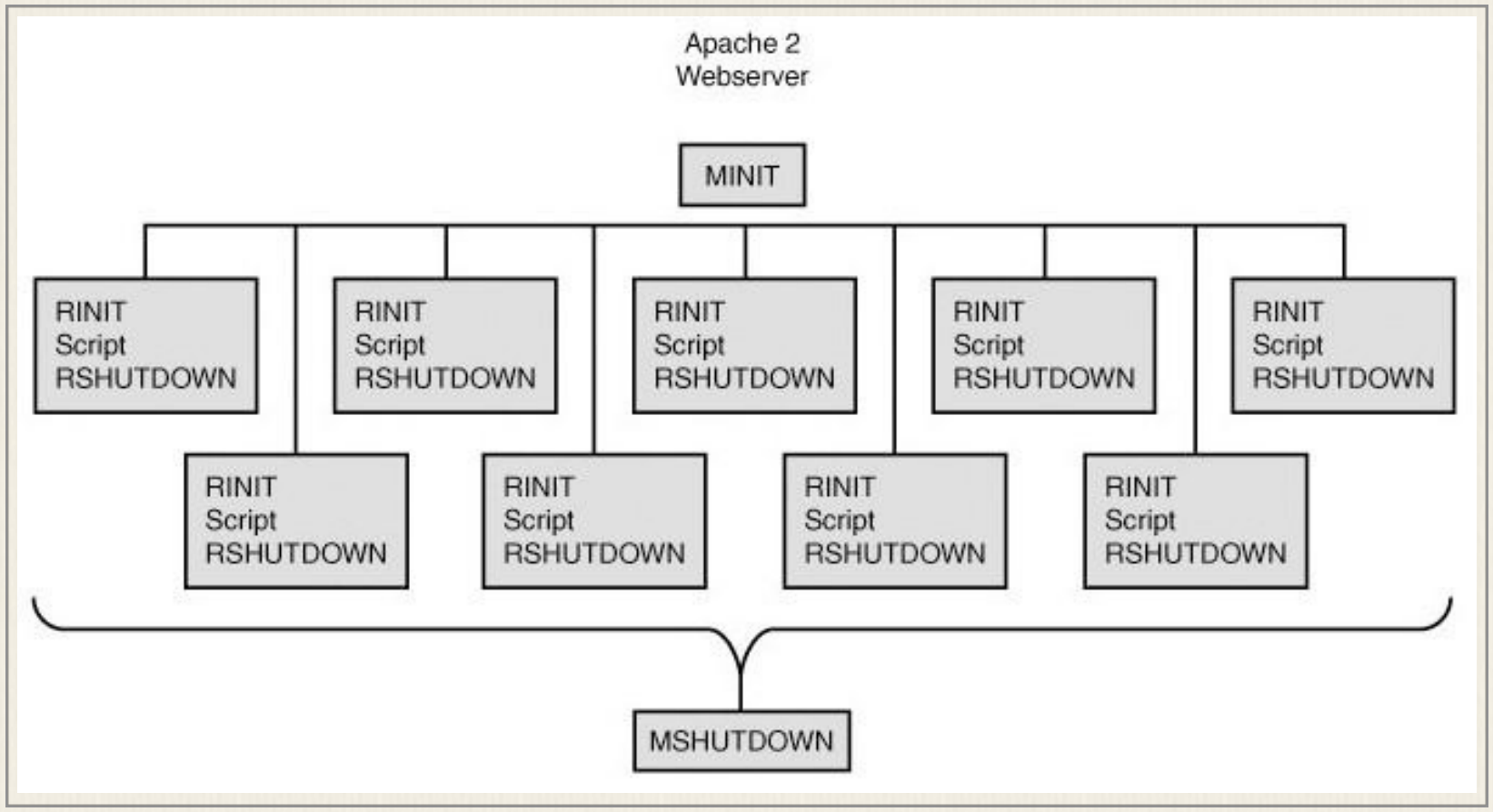


图5. 多线程生命周期

4. FastCGI模式

在我们用的Nginx+PHP-FPM 用的就是FastCGI模式，Fastcgi是一种特殊的CGI模式，是一种常驻进 程类型的CGI，运行后可以Fork多个进程，不用花费时间动态的Fork子进程，也不需要每次请求都调用MINT/MSHUTDOWN。PHP通过 PHP-FPM 来管理和调度FastCGI的进程池。Nginx和PHP-FPM通过本地的TCP Socket和Unix Socket 进行通信。

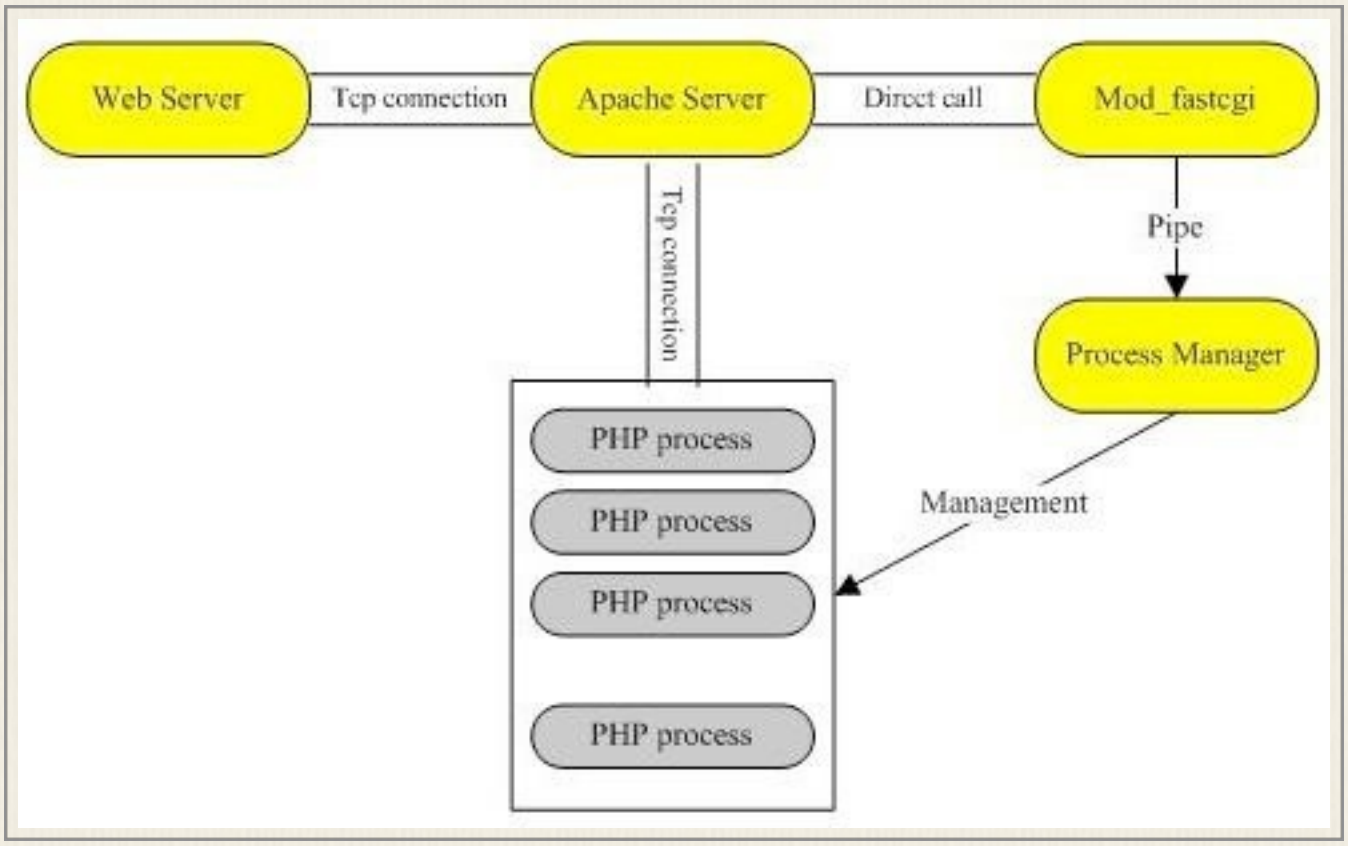


图6. FastCGI模式生命周期

PHP-FPM进程管理器自身初始化，启动多个CGI解释器进程等待来自Nginx的请求。当客户端请求达到PHP-FPM，管理器选择到一个 CGI进程进行处理，Nginx将CGI环境变量和标准输入发送到一个PHP-CIG子进程。PHP-CGI子进程处理完成后，将标准输出和错误信息返回 给Nginx，当PHP-CGI子进程关闭连接时，请求处理完成。PHP-CGI子进程等待着下一个连接。

可以想象CGI的系统开销有多大。每一个Web 请求PHP都必须重新解析php.ini、载入全部扩展并始化全部数据结构。使用FastCGI，所有这些都只在进程启动时发生一次。另外，对于数据库和Memcache的持续连接可以工作。

5. 内嵌模式（Embedded）

Embed SAPI是一种特殊的SAPI，允许在C/C++ 语言中调用PHP提供的函数。这种SAPI和CLI模式一样，按照Module Init => Request Init => Request => Request Shutdown => Module Shutdown的模式运行。

Embed SAPI可以调用PHP丰富的类库，也可以实现高级玩法，比如可以查看PHP的OPCODE（PHP执行的中间码，Zend引擎的指令，由PHP代码生成）。

详细请见： http://www.laruence.com/2008/09/23/539.html

SAPI的运行机制

我们以CGI为例，看一下SAPI的运行机制。

```
1.  static sapi_module_struct cgi_sapi_module = {
2.      "cgi-fcgi",                 /* 输出给php_info()使用 */    "CGI/FastCGI",              /* pretty name */
3.      php_cgi_startup,            /* startup 当SAPI初始化时，首先会调用该函数 */
4.      php_module_shutdown_wrapper,    /* shutdown 关闭函数包装器，它用来释放所有的SAPI的数据结构、内存等，调用php_module_shutdown */
5.      sapi_cgi_activate,          /* activate 此函数会在每个请求开始时调用，它会做初始化，资源分配 */
6.      sapi_cgi_deactivate,        /* deactivate 此函数会在每个请求结束时调用，它用来确保所有的数据都得到释放 */
7.      sapi_cgi_ub_write,          /* unbuffered write 不缓存的写操作(unbuffered write)，它是用来向SAPI外部输出数据 */
8.      sapi_cgi_flush,             /* flush 刷新输出，在CLI模式下通过使用C语言的库函数fflush实现*/    NULL,                       /* get uid */
9.      sapi_cgi_getenv,            /* getenv 根据name查找环境变量 */
```

```
10.     php_error,                  /* error handler 注册错误处理函数 */
11.     NULL,                       /* header handler PHP调用header()时候被调用 */
12.     sapi_cgi_send_headers,      /* send headers handler 发送头部信息*/
13.     NULL,                       /* send header handler 发送一个单独的头部信息 */
14.     sapi_cgi_read_post,         /* read POST data  当请求的方法是POST时，程序获取POST数据，写入$_POST数组 */
15.     sapi_cgi_read_cookies,      /* read Cookies 获取Cookie值 */
16.     sapi_cgi_register_variables,    /* register server variables 给$_SERVER添加环境变量 */
17.     sapi_cgi_log_message,       /* Log message 输出错误信息 */
18.     NULL,                       /* Get request time */
19.     NULL,                       /* Child terminate */
20.     STANDARD_SAPI_MODULE_PROPERTIES
21. };
```

由上面代码可见，PHP的SAPI像是面向对象中基类，SAPI.h和SAPI.c包含的函数是抽象基类的声明和定义，各个服务器用的SAPI模式，则是继承了这个基类，并重新定义基类方法的子类。

总结

PHP的SAPI是Zend引擎提供的一组标准交互接口，通过注册初始化、析构、输入、输出等接口，我们可以将应用程序运行在Zend引擎上，也可以

把PHP嵌入到类似Apache的Web Server中。PHP常见的SAPI模式有五种，CGI/CLI模式、多进程模式、多线程模式、FastCGI模式和内嵌模式。

了解PHP的SAPI机制意义重大，帮助我们理解PHP的生命周期，并了解如何更好的通过C/C++为PHP编写扩展，并在生命周期中找到提高系统性能的方式。

原文链接：http://www.csdn.net/article/2014-09-26/2821885-exploring-of-the-php-2

用 C 语言编写一个简单的垃圾回收器

译者：forever_you

人们似乎认为编写垃圾回收机制是很难的，是一种只有少数智者和Hans Boehm(et al)才能理解的高深魔法。我认为编写垃圾回收最难的地方就是内存分配，这和阅读K&R所写的malloc样例难度是相当的。

在开始之前有一些重要的事情需要说明一下：第一，我们所写的代码是基于Linux Kernel的，注意是Linux Kernel而不是GNU/Linux。第二，我们的代码是32bit的。第三，请不要直接使用这些代码。我并不保证这些代码完全正确，可能其中有一些我 还未发现的小的bug，但是整体思路仍然是正确的。好了，让我们开始吧。

如果你看到任何有误的地方，请邮件联系我maplant2@illinois.edu

编写malloc

最开始，我们需要写一个内存分配器(memmory allocator)，也可以叫做内存分配函数(malloc function)。最简单的内存分配实现方法就是维护一个由空闲内存块组成的链表，这些空闲内存块在需要的时候被分割或分配。当用户请求一块内存时，一 块合适大小的内存块就会从链表中被移除并分配给用户。如果链表中没有合适的空闲内存块存在，而且更大的空闲内存块已经被分割成小的内存块了或内核也正在请 求更多的内存(译者注：就是链表中的空闲内存块都太小不足以分配给用户的情况)。那么此时，会释放掉一块内存并把它添加到空闲块链表中。

在链表中的每个空闲内存块都有一个头(header)用来描述内存块的信息。我们的header包含两个部分，第一部分表示内存块的大小，第二部分指向下一个空闲内存块。

```
typedef struct header{
    unsigned int  size;
    struct block  *next;
} header_t;
```

将头(header)内嵌进内存块中是唯一明智的做法，而且这样还可以享有字节自动对齐的好处，这很重要。

由于我们需要同时跟踪我们“当前使用过的内存块”和“未使用的内存块”，因此除了维护空闲内存的链表外，我们还需要一条维护当前已用内存块的链表 (为了方便，这两条链表后面分别写为“空闲块链表”和“已用块链表”)。我们从空闲块链表中移除的内存块会被添加到已用块链表中，反之亦然。

现在我们差不多已经做好准备来完成malloc实现的第一步了。但是再那之前，我们需要知道怎样向内核申请内存。

动态分配的内存会驻留在一个叫做堆(heap)的地方，堆是介于栈(stack)和BSS(未初始化的数据段 – 你所有的全局变量都存放在这里且具有 默认值为0)之间的一块内存。堆(heap)的内存地址起始于(低地址)BSS段的边界，结束于一个分隔地址(这个分隔地址是已建立映射的内存和未建立映 射的内存的分隔线)。为了能够从内核中获取更多的内存，我们只需提高这个分隔地址。为了提高这个分隔地址我们需要调用一个叫作 sbrk 的Unix系统的系统调用，这个函数可以根据我们提供的参数来提高分隔地址，如果函数执行成功则会返回以前的分隔地址，如果失败将会返回 – 1。

利用我们现在知道的知识，我们可以创建两个函数：morecore()和add_to_free_list()。当空闲块链表缺少内存块时，我们调 用morecore()函数来申请更多的内存。由于每次向内核申请内存的代价是昂贵的，我们以页(page-size)为单位申请内存。页的大小在这并不 是很重要的知识点，不过这有一个很简单解释：页是虚拟内存映射到物理内存的最小内存单位。接下来我们就可以使用add_to_list()将申请到的内存 块加入空闲块链表。

```
    /*
     * Scan the free list and look for a place to put the block. Basically, we're
     * looking for any block the to be freed block might have been partitioned from.
     */
    static void
    add_to_free_list(header_t *bp)
    {
        header_t *p;

        for (p = freep; !(bp > p && bp < p->next); p = p->next)
            if (p >= p->next && (bp > p || bp < p->next))
                break;

    if (bp + bp->size == p->next) {
        bp->size += p->next->size;
        bp->next = p->next->next;
    } else
        bp->next = p->next;

    if (p + p->size == bp) {
        p->size += bp->size;
        p->next = bp->next;
    } else
```

```
        p->next = bp;

    freep = p;
}

#define MIN_ALLOC_SIZE 4096 /* We allocate blocks in page sized
chunks. */

/*
 * Request more memory from the kernel.
 */
static header_t *
morecore(size_t num_units)
{
    void *vp;
    header_t *up;

    if (num_units < MIN_ALLOC_SIZE)
        num_units = MIN_ALLOC_SIZE / sizeof(header_t);

    if ((vp = sbrk(num_units * sizeof(header_t))) == (void *) -1)
        return NULL;

    up = (header_t *) vp;
```

```
    up->size = num_units;
    add_to_free_list (up);
    return freep;
}
```

现在我们有了两个有力的函数，接下来我们就可以直接编写malloc函数了。我们扫描空闲块链表当遇到第一块满足要求的内存块(内存块比所需内存大 即满足要求)时，停止扫描，而不是扫描整个链表来寻找大小最合适的内存块，我们所采用的这种算法思想其实就是首次适应(与最佳适应相对)。

注意：有件事情需要说明一下，内存块头部结构中size这一部分的计数单位是块(Block)，而不是Byte。

```
static header_t base; /* Zero sized block to get us started. */
static header_t *usedp, *freep;

/*
 * Find a chunk from the free list and put it in the used list.
 */
void *
GC_malloc(size_t alloc_size)
{
    size_t num_units;
    header_t *p, *prevp;
```

```
        num_units = (alloc_size + sizeof(header_t) - 1) / sizeof(header_t) +
1;
        prevp = freep;

        for (p = prevp->next;; prevp = p, p = p->next) {
            if (p->size >= num_units) { /* Big enough. */
                if (p->size == num_units) /* Exact size. */
                    prevp->next = p->next;
                else {
                    p->size -= num_units;
                    p += p->size;
                    p->size = num_units;
                }
```

```
freep = prevp;

/* Add to p to the used list. */
if (usedp == NULL)
    usedp = p->next = p;
else {
    p->next = usedp->next;
    usedp->next = p;
}
```

```
            return (void *) (p + 1);
        }
        if (p == freep) { /* Not enough memory. */
            p = morecore(num_units);
            if (p == NULL) /* Request for more memory failed. */
                return NULL;
        }
    }
}
```

注意这个函数的成功与否，取决于我们第一次使用时是否使 freep = &base 。这点我们会在初始化函数中进行设置。

尽管我们的代码完全没有考虑到内存碎片，但是它能工作。既然它可以工作，我们就可以开始下一个有趣的部分 – 垃圾回收！

标记和清扫

我们说过垃圾回收器会很简单，因此我们尽可能的使用简单的方法：标记和清除方式。这个算法分为两个部分：

首先，我们需要扫描所有可能存在指向堆中数据(heap data)的变量的内存空间并确认这些内存空间中的变量是否指向堆中的数据。为了做到这点，对于可能内存空间中的每个字长(word-size)的数据 块，我们遍历已用块链表中的内存块。如果数据块所指向的内存是在已用链表块中的某一内存块中，我们对这个内存块进行标记。

第二部分是，当扫描完所有可能的内存空间后，我们遍历已用块链表将所有未被标记的内存块移到空闲块链表中。

现在很多人会开始认为只是靠编写类似于malloc那样的简单函数来实现C的垃圾回收是不可行的，因为在函数中我们无法获得其外面的很多信息。例如，在C语言中没有函数可以返回分配到堆栈中的所有变量的哈希映射。但是只要我们意识到两个重要的事实，我们就可以绕过这些东西：

第一，在C中，你可以尝试访问任何你想访问的内存地址。因为不可能有一个数据块编译器可以访问但是其地址却不能被表示成一个可以赋值给指针的整数。 如果一块内存在C程序中被使用了，那么它一定可以被这个程序访问。这是一个令不熟悉C的编程者很困惑的概念，因为很多编程语言都会限制程序访问虚拟内存， 但是C不会。

第二，所有的变量都存储在内存的某个地方。这意味着如果我们可以知道变量们的通常存储位置，我们可以遍历这些内存位置来寻找每个变量的所有可能值。另外，因为内存的访问通常是字(word-size)对齐的，因此我们仅需要遍历内存区域中的每个字(word)即可。

局部变量也可以被存储在寄存器中，但是我们并不需要担心这些因为寄存器经常会用于存储局部变量，而且当函数被调用的时候他们通常会被存储在堆栈中。

现在我们有一个标记阶段的策略：遍历一系列的内存区域并查看是否有内存可能指向已用块链表。编写这样的一个函数非常的简洁明了：

```
#define UNTAG(p) (((unsigned int) (p)) & 0xfffffffc)


/*
 * Scan a region of memory and mark any items in the used list appropri-
ately.
 * Both arguments should be word aligned.
 */
static void
mark_from_region(unsigned int *sp, unsigned int *end)
{
    header_t *bp;
```

```
    for (; sp < end; sp++) {
        unsigned int v = *sp;
        bp = usedp;
        do {
            if (bp + 1 <= v &&
                bp + 1 + bp->size > v) {
                bp->next = ((unsigned int) bp->next) | 1;
                break;
            }
        } while ((bp = UNTAG(bp->next)) != usedp);
    }
}
```

为了确保我们只使用头(header)中的两个字长(two words)我们使用一种叫做标记指针(tagged pointer)的技术。利用header中的next指针指向的地址总是字对齐(word aligned)这一特点，我们可以得出指针低位的几个有效位总会是0。因此我们将next指针的最低位进行标记来表示当前块是否被标记。

现在，我们可以扫描内存区域了，但是我们应该扫描哪些内存区域呢？我们要扫描的有以下这些：

1. BBS(未初始化数据段)和初始化数据段。这里包含了程序的全局变量和局部变量。因为他们有可能应用堆(heap)中的一些东西，所以我们需要扫描BSS与初始化数据段。

2. 已用的数据块。当然，如果用户分配一个指针来指向另一个已经被分配的内存块，我们不会想去释放掉那个被指向的内存块。

3. 堆栈。因为堆栈中包含所有的局部变量，因此这可以说是最需要扫描的区域了。

我们已经了解了关于堆(heap)的一切，因此编写一个mark_from_heap函数将会非常简单：

```
/*
 * Scan the marked blocks for references to other unmarked blocks.
 */
static void
mark_from_heap(void)
{
    unsigned int *vp;
    header_t *bp, *up;

    for (bp = UNTAG(usedp->next); bp != usedp; bp = UNTAG(bp->next))
{
        if (!((unsigned int)bp->next & 1))
            continue;
        for (vp = (unsigned int *)(bp + 1);
             vp < (bp + bp->size + 1);
             vp++) {
            unsigned int v = *vp;
            up = UNTAG(bp->next);
            do {
                if (up != bp &&
                    up + 1 <= v &&
                    up + 1 + up->size > v) {
```

```
                up->next = ((unsigned int) up->next) | 1;
                break;
            }
        } while ((up = UNTAG(up->next)) != bp);
    }
  }
}
```

幸运的是对于BSS段和已初始化数据段，大部分的现代u-nix链接器可以导出 etext 和 end 符号。etext 符号的地址是初始化数据段的起点(the last address past the text segment，这个段中包含了程序的机器码)，end符号是堆(heap)的起点。因此，BSS和已初始化数据段位于 &etext 与 &end 之间。这个方法足够简单，当不是平台独立的。

堆栈这部分有一点困难。堆栈的栈顶非常容易找到，只需要使用一点内联汇编即可，因为它存储在 sp 这个寄存器中。但是我们将会使用的是 bp 这个寄存器，因为它忽略了一些局部变量。

寻找堆栈的的栈底(堆栈的起点)涉及到一些技巧。出于安全因素的考虑，内核倾向于将堆栈的起点随机化，因此我们很难得到一个地址。老实说，我在寻找 栈底方面并不是专家，但是我有一些点子可以帮你找到一个准确的地址。一个可能的方法是，你可以扫描调用栈(call stack)来寻找 env 指针，这个指针会被作为一个参数传递给主程序。另一种方法是从栈顶开始读取每个更大的后续地址并处理inexorible SIGSEGV。但是我们并不打算采用这两种方法中的任何一种，我们将利用linux会将栈底放入一个字符串并存于proc目录下表示该进程的文件中这一 事实。这听起来很愚蠢而且非常间接。值得庆幸的是，我并不感觉这样做是滑稽的，因为它和Boehm GC中寻找栈底所用的方法完全相同。

现在我们可以编写一个简单的初始化函数。在函数中，我们打开proc文件并找到栈底。栈底是文件中第28个值，因此我们忽略前27个值。Boehm

GC和我们的做法不同的是他仅使用系统调用来读取文件来避免让stdlib库使用堆(heap)，但是我们并不在意这些。

```
/*
 * Find the absolute bottom of the stack and set stuff up.
 */
void
GC_init(void)
{
    static int initted;
    FILE *statfp;

    if (initted)
    return;

initted = 1;

statfp = fopen("/proc/self/stat", "r");
assert(statfp != NULL);
fscanf(statfp,
        "%*d %*s %*c %*d %*d %*d %*d %*d %*u "
        "%*lu %*lu %*lu %*lu %*lu %*lu %*ld %*ld "
        "%*ld %*ld %*ld %*ld %*llu %*lu %*ld "
        "%*lu %*lu %*lu %lu", &stack_bottom);
```

```
    fclose(statfp);


    usedp = NULL;
    base.next = freep = &base;
    base.size = 0;
```

现在我们知道了每个我们需要扫描的内存区域的位置，所以我们终于可以编写显示调用的回收函数了：

```
/*
 * Mark blocks of memory in use and free the ones not in use.
 */
void
GC_collect(void)
{
    header_t *p, *prevp, *tp;
    unsigned long stack_top;
    extern char end, etext; /* Provided by the linker. */

    if (usedp == NULL)
        return;

    /* Scan the BSS and initialized data segments. */
    mark_from_region(&etext, &end);
```

```
        /* Scan the stack. */
        asm volatile ("movl    %%ebp, %0" : "=r" (stack_top));
        mark_from_region(stack_top, stack_bottom);

        /* Mark from the heap. */
        mark_from_heap();

        /* And now we collect! */
        for (prevp = usedp, p = UNTAG(usedp->next);; prevp = p, p =
UNTAG(p->next)) {
        next_chunk:
            if (!((unsigned int)p->next & 1)) {
                /*
                 * The chunk hasn't been marked. Thus, it must be set free.
                 */
                tp = p;
                p = UNTAG(p->next);
                add_to_free_list(tp);

                if (usedp == tp) {
                    usedp = NULL;
                    break;
                }
```

```
            prevp->next = (unsigned int)p | ((unsigned int) prevp->next & 1);
            goto next_chunk;
        }
        p->next = ((unsigned int) p->next) & ~1;
        if (p == usedp)
            break;
    }
}
```

朋友们，所有的东西都已经在这了，一个用C为C程序编写的垃圾回收器。这些代码自身并不是完整的，它还需要一些微调来使它可以正常工作，但是大部分代码是可以独立工作的。

总结

从小学到高中，我一直在学习打鼓。每个星期三的下午4:30左右我都会更一个很棒的老师上打鼓教学课。

每当我在学习一些新的打槽（groove）或节拍时，我的老师总会给我一个相同的告诫：我试图同时做所有的事情。我看着乐谱，我只是简单地尝试用双 手将它全部演奏出来，但是我做不到。原因是因为我还不知道怎样打槽，但我却在学习打槽地时候同时学习其它东西而不是单纯地练习打槽。

因此我的老师教导我该如何去学习：不要想着可以同时做所有地事情。先学习用你地右手打架子鼓，当你学会之后，再学习用你的左手打小鼓。用同样地方式 学习贝斯、手鼓和其它部分。当你可以单独使用每个部分之后，慢慢开始同时练习它们，先两个同时练习，然后三个，最后你将可以可以同时完成所有部分。

我在打鼓方面从来都不够优秀，但我在编程时始终记着这门课地教训。一开始就打算编写完整的程序是很困难的，你编程的唯一算法就是分而治之。先编写内存分配函数，然后编写查询内存的函数，然后是清除内存的函数。最后将它们合在一起。

当你在编程方面克服这个障碍后，就再也没有困难的实践了。你可能有一个算法不太了解，但是任何人只要有足够的时间就肯定可以通过论文或书理解这个算 法。如果有一个项目看起来令人生畏，那么将它分成完全独立的几个部分。你可能不懂如何编写一个解释器，但你绝对可以编写一个分析器，然后看一下你还有什么 需要添加的，添上它。

原文链接：http://blog.jobbole.com/77248/

[MySQL优化案例]系列 — 索引、提交频率对InnoDB表写入速度的影响

作者：叶金荣

本次，我们来看看索引、提交频率对InnoDB表写入速度的影响，了解有哪些需要注意的。

先直接说几个结论吧：

1、关于索引对写入速度的影响：

a、如果有自增列做主键，相对完全没索引的情况，写入速度约提升3.11%；

b、如果有自增列做主键，并且二级索引，相对完全没索引的情况，写入速度约降低 27.37%；

因此，InnoDB表最好总是有一个自增列做主键。

2、关于提交频率对写入速度的影响（以表中只有自增列做主键的场景，一次写入数据30万行数据为例）：

a、等待全部数据写入完成后，最后再执行commit提交的效率最高；

b、每10万行提交一次，相对一次性提交，约慢了1.17%；

c、每1万行提交一次，相对一次性提交，约慢了3.01%；

d、每1千行提交一次，相对一次性提交，约慢了23.38%；

e、每100行提交一次，相对一次性提交，约慢了24.44%；

f、每10行提交一次，相对一次性提交，约慢了92.78%；

g、每行提交一次，相对一次性提交，约慢了546.78%，也就是慢了5倍；

因此，最好是等待所有事务结束后再批量提交，而不是每执行完一个SQL就提交一次。

曾经有一次对比测试mysqldump启用extended-insert和未启用导出的SQL脚本，后者比前者慢了不止5倍。

下面是详细的测试案例过程，有兴趣的同学可以看看：

```
DROP TABLE IF EXISTS `mytab`;
CREATE TABLE `mytab` (
`id` int(10) unsigned NOT NULL AUTO_INCREMENT,
`c1` int(11) NOT NULL DEFAULT '0',
`c2` int(11) NOT NULL DEFAULT '0',
`c3` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
`c4` varchar(200) NOT NULL DEFAULT '',
PRIMARY KEY (`id`)
) ENGINE=InnoDB;
DELIMITER $$$
DROP PROCEDURE IF EXISTS `insert_mytab`;
CREATE PROCEDURE `insert_mytab`(in rownum int, in commitrate int)
BEGIN
DECLARE i INT DEFAULT 0;
SET AUTOCOMMIT = 0;
WHILE i < rownum DO INSERT INTO mytab(c1, c2, c3,c4) VALUES(FLOOR(RAND()*rownum),FLOOR(RAND()*rownum),NOW(), REPEAT(CHAR(ROUND(RAND()*255)),200)); SET i = i+1; /* 达到每 COMMITRATE 频率时提交一次 */ IF (commitrate > 0) AND (i % commitrate = 0) THEN
COMMIT;
SELECT CONCAT('commitrate: ', commitrate, ' in ', I);
END IF;
```

```
    END WHILE;
    /* 最终再提交一次,确保成功 */
COMMIT;
SELECT 'ALL COMMIT;';
    END; $$$
    #测试调用
call insert_mytab(300000, 1); — 每次一提交
call insert_mytab(300000, 10); — 每10次一提交
call insert_mytab(300000, 100); — 每100次一提交
call insert_mytab(300000, 1000); — 每1千次一提交
call insert_mytab(300000, 10000); — 每1万次提交
call insert_mytab(300000, 100000); — 每10万次一提交
call insert_mytab(300000, 0); — 一次性提交
```

测试耗时结果对比：

单位（秒）	主键+普通索引	完全无索引	只有主键
每行提交	99.26189	107.407858	108.653024
每10行提交	53.260255	32.015148	32.240613
每100行提交	48.301461	20.664805	19.432488
每1千行提交	24.001181	20.489308	20.19498
每1万行提交	20.821781	17.106163	19.310056
每10万行提交	20.62335	16.801367	17.029532
一次性提交	21.152014	16.606677	16.10582

原文链接：http://imysql.com/2014/09/24/mysql-optimization-case-how-index-and-commit-rate-affect-innodb-insert.shtml

bash代码注入的安全漏洞

作者：陈皓

很多人或许对上半年发生的安全问题“心脏流血”（Heartbleed Bug）事件记忆颇深，这两天，又出现了另外一个“毁灭级”的漏洞——Bash软件安全漏洞。这个漏洞由法国GNU/Linux爱好者Stéphane Chazelas所发现。随后，美国电脑紧急应变中心（US-CERT）、红帽以及多家从事安全的公司于周三（北京时间9月24日）发出警告。 关于这个安全漏洞的细节可参看美国政府计算安全的这两个漏洞披露：CVE-2014-6271 和 CVE-2014-7169。

这个漏洞其实是非常经典的“注入式攻击”，也就是可以向 bash注入一段命令，从bash1.14 到4.3都存在这样的漏洞。我们先来看一下这个安全问题的症状。

Shellshock (CVE-2014-6271)

下面是一个简单的测试：

```
$ env VAR='() { :;}; echo Bash is vulnerable!' bash -c "echo Bash Test"
```

如果你发现上面这个命令在你的bash下有这样的输出，那你就说明你的bash是有漏洞的：

```
Bash is vulnerable!
Bash Test
```

简单地看一下，其实就是向环境变量中注入了一段代码 echo Bash is vulnerable。关于其中的原理我会在后面给出。

很快，CVE-2014-6271的官方补丁出来的了——Bash-4.3 Official Patch 25。

AfterShock – CVE-2014-7169 （又叫Incomplete fix to Shellshock）

但随后，马上有人在Twitter上发贴——说这是一个不完整的fix，并给出了相关的攻击方法。

也就是下面这段测试代码（注意，其中的sh在linux下等价于bash）：

```
env X='() { (a)=>\' sh -c "echo date"; cat echo
```

上面这段代码运行起来会报错，但是它要的就是报错，报错后会在你在当前目录下生成一个echo的文件，这个文件的内容是一个时间文本。下面是上面 这段命令执行出来的样子。

```
$ env X='() { (a)=>\' sh -c "echo date"; cat echo
sh: X: line 1: syntax error near unexpected token `='
sh: X: line 1: `'
sh: error importing function definition for `X'
Sat Sep 27 22:06:29 CST 2014
```

这段测试脚本代码相当的诡异，就像“天书”一样，我会在后面详细说明这段代码的原理。

原理和技术细节

要说清楚这个原理和细节，我们需要从 bash的环境变量开始说起。

bash的环境变量

环境变量大家知道吧，这个不用我普及了吧。环境变量是操作系统运行shell中的变量，很多程序会通过环境变量改变自己的执行行为。在bash中要定义一个环境变量的语法很简单（注：=号的前后不能有空格）：

```
$ var="hello world"
```

然后你就可以使用这个变量了，比如：echo $var什么的。但是，我们要知道，这个变量只是一个当前shell的“局部变量”，只在当前的shell进程中可以访问，这个shell进程fork出来的进程是访问不到的。

你可以做这样的测试：

```
$ var="hello coolshell"
$ echo $var
hello coolshell
$ bash
```

```
$ echo $var
```

上面的测试中，第三个命令执行了一个bash，也就是开了一个bash的子进程，你就会发现var不能访问了。

为了要让shell的子进程可以访问，我们需要export一下：

```
$ export var="hello coolshell"
```

这样，这个环境变量就会在其子进程中可见了。

如果你要查看一下有哪些环境变量可以在子进程中可见（也就是是否被export了），你可使用env命令。不过，env命令也可以用来定义export的环境变量。如下所示：

```
$ env $var="hello haoel"
```

有了这些基础知识还不够，我们还要知道一个基础知识——shell的函数。

bash的函数

在bash下定义一个函数很简单，如下所示：

```
$ foo(){ echo "hello coolshell"; }
$ foo
hello coolshell
```

有了上面的环境变量的基础知识后，你一定会想试试这个函数是否可以在子进程中调用，答案当然是不行的。

```
$ foo(){ echo "hello coolshell"; }
$ foo
hello coolshell
$ bash
$ foo
bash: foo: command not found
```

你看，和环境变量是一样的，如果要在子进程中可以访问的话，那么，还是一样的，需要export，export有个参数 -f，意思是export一个函数。如：

```
$ foo(){ echo "hello coolshell"; }
$ foo
hello coolshell
$ export -f foo
$ bash
$ foo
hello coolshell
```

好了，我讲了这么半天的基础知识，别烦，懂了这些，你才会很容易地理解这两个漏洞是怎么回事。

好，现在要进入正题。

bash的bug

从上面我们可以看到，bash的变量和函数用了一模一样的机制，如果你用env命令看一下export出来的东西，你会看到上面我们定义的变量和函数都在，如下所示（我省略了其它的环境变量）：

```
$ env
var=hello coolshell
foo=() { echo "hello coolshell"
}
```

原来，都用同样的方式啊——无论是函数还是变量都是变量啊。于是，看都不用看bash的源代码，聪明的黑客就能猜得到——bash判断一个环境变量是不是一个函数，就看它的值是否以”()”开始。于是，一股邪念涌上心头。

黑客定义了这样的环境变量（注：() 和 { 间的空格不能少）：

```
$ export X='() { echo "inside X"; }; echo "outside X";'
```

env一下，你会看到X已经在了：

```
$ env
X=(){ echo "inside X"; }; echo "outside X";
```

然后，当我们在当前的bash shell进程下产生一个bash的子进程时，新的子进程会读取父进程的所有export的环境变量，并复制到自己的进程空间中，很明显，上面的X变量的 函数的后面还注入了一条命令：echo “outside

X”，这会在父进程向子进程复制的过程中被执行吗？（关于fork相关的东西你可以看一下我以前写的《fork的一个面试题》）

答案是肯定的。

$ export X='() { echo "inside X"; }; echo "outside X";'

$ bash

outside X

你看，一个代码注入就这样完成了。这就是bash的bug—— 函数体外面的代码被默认地执行了。

我们并不一定非要像上面那样创建另一个bash的子进程，我们可以使用bash -c的参数来执行一个bash子进程命令。就像这个安全漏洞的测试脚本一样：

env VAR='() { :;}; echo Bash is vulnerable!' bash -c "echo Bash Test"

其中，() { :;} 中的冒号就相当于/bin/true，返回true并退出。而bash -c其实就是在spawn一个bash的echo的子进程，用于触发函数体外的echo命令。所以，更为友好一点的测试脚本应该是：

env VAR='() { :;}; echo Bash is vulnerable!' bash -c "echo 如果你看到了vulnerable字样说明你的bash有安全问题"

OK，你应该明白这个漏洞是怎么一回事了吧。

bash漏洞的影响有多大

在网上看到好多人说这个漏洞不大，还说这个事只有那些陈旧的执行CGI脚本的网站才会有，现在已经没有网站用CGI了。我靠，这真是无知者无畏啊。

我举个例子，如果你的网站中有调用操作系统的shell命令，比如你用PHP执行个exec之类的东西。这样的需求是有的，特别是对于一些需要和操作系统交互的重要的后台用于系统管理的程序。于是就会开一个bash的进程来执行。

我们还知道，现在的HTTP服务器基本上都是以子进程式的，所以，其中必然会存在export 一些环境变量的事，而有的环境变量的值是从用户端来的，比如：HTTP_USER_AGENT这样的环境变量，只由浏览器发出的。其实这个变量你想写成什么就写成什么。

于是，我可以把这个HTTP_USER_AGENT的环境变量设置成上述的测试脚本，只不过，我会把echo Bash is vulnerable!这个东西换成别的更为凶残的命令。呵呵。

关于这个漏洞会影响哪些已有的系统，你可以自己Google，几乎所有的报告这个漏洞的文章都说了（比如：这篇，这篇），我这里就不复述了。

注：如果你要看看你的网站有没有这样的问题，你可以用这个在线工具测试一下：‘ShellShock’ Bash Vulnerability CVE-2014-6271 Test Tool。

现在，你知道这事可能会很大了吧。还不赶快去打补丁。（注，yum update bash 把bash版本升级到 4.1.2-15.el6_5.2 ， ）

关于 AfterShock – CVE-2014-7169 测试脚本的解释

很多同学没有看懂下面这个测试脚本是什么意思，我这里解释一下。

```
env X='() { (a)=>\' sh -c "echo date"; cat echo
```

- X='() { (a)=>\' 这个不用说了，定义一个X的环境变量。但是，这个函数不完整啊，是的，这是故意的。另外你一定要注意，\'不是为了单引号的转义，X这个变量的值就是 () { (a)=>\

- 其中的 (a)=这个东西目的就是为了让bash的解释器出错（语法错误）。

- 语法出错后，在缓冲区中就会只剩下了 “>\”这两个字符。

- 于是，这个神奇的bash会把后面的命令echo date换个行放到这个缓冲区中，然后执行。

相当于在shell 下执行了下面这个命令：

```
$ >\
echo date
```

如果你了解bash，你会知道 \ 是用于命令行上换行的，于是相当于执行了：

```
$ >echo date
```

这不就是一个重定向么？上述的命令相当于：

```
$ date > echo
```

于是，你的当前目录下会出现一个echo的文件，这个文件的内容就是date命令的输出。

能发现这个种玩法的人真是个变态，完全是为bash的源代码量身定制的一个攻击。

原文链接：http://coolshell.cn/articles/11973.html

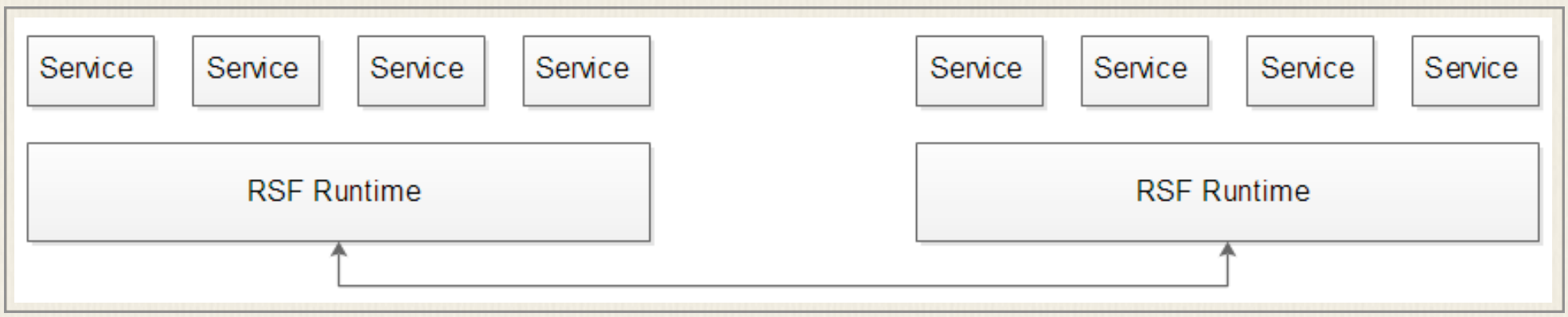
RSF 分布式服务框架设计

作者：哈库纳

是时候设计一个分布式服务框架了。我先将它定名为 Hasor-RSF，“RSF”为 Remote Service Framework 的缩写。

RSF的目的是为了提供一种高效的远程服务访问方式，例如“A机器访问在B机器上的一个服务”。当然首先它是运行在Java上的，但是我并不希望 Java 成为 RSF的唯一平台。

它应该是分布式的，就是说服务 A 可能会分布在若干台机器内。 当我的应用打算调用这个服务时我应该可以在这若干服务提供的机器上随机调用。这样做的好处是有助于高并发、高访问、高可用。



RSF 的本质其实就是 RPC 那么我们可以先对比一下 RPC 里都有什么可以被我们拿来选用。下面列出来的只是其中一些我相信聪明的朋友们会列举出更多的解决方案，我也敢保证你们知道的比我还多。

1. Java原生的 RMI。
2. Hessian
3. WebServices

4. Restful
5. HTTP Request
6. RTMP/AMF
7. 淘宝的 HSF、Dubbo

RMI，这个 Java 原生的东东似乎从一开始就没有被人们所看好，究其原因是速度太慢。但是它的好处是Java原生，使用 RMI 不需要引入其它任何第三方软件包。不过挑剔的同学们似乎不太看好这个优点。

Hessian，原则上说Hessian我并不认为它是一个远程服务框架范畴的东西。我更觉得 Hessian 是一种数据交互格式。就像是 JSON，XML-RPC，AMF，Kryo 一类的东西。Hessian 的优点是大量的兼容平台例如：“IOS、Java、.net、C++、Python、Flash、Ruby、PHP”，其次它的第二个有点是二进制格式。在大对象序列化上会占有很大的优势。

WebServices，一个老牌技术解决方案。在我印象中 WebServices 是跟随着 SOA 这个东西一起出名的，他有一个最大的好处是防火墙穿透。毕竟人家是靠 80 端口吃饭的，牛叉的很。不过话说回来WebServices的最大要害就是，Xml传输格式。把一个对象序列化成为一个Xml数据是一件很容易的事，但是 反序列化成本似乎是很高。再加上 SOAP 协议本身是建立在 XML 形式上，这就使得 Web Service 奇慢无比了。当然因素还有很多我就不多说了。

Restful，其实 restful 我更觉得它是一种 API 表述规范。但在社区论坛中讨论看来，restful 的应用似乎也延伸到远程服务的领域。所以有必要说明一下。restful 最初是出现在 web 上，究其本质是还是 HTTP。例如对于：“http://xxxxx/xxxx”这个资源的访问可以利用 HTTP 的“GET、PUT、DELETE”等方法对资源操作加以描述说明。我个人觉得这东西用在 RPC 上并不合适。

HTTP，这是我用过最多的一种远程交互方式。远离很见dna，服务发布者将服务发布成为一个http资源。调用者请求这个http资源。数据传输格式完 全程序双方自行协商。这种方法简单除暴行之有效。不过缺点是我们要自己补充通信协议，例如请求参数和响应数据格式。常规的交互格式有JSON、XML。

RTMP/AMF ，这个组合的确是一套很完善的远程调用解决方案。RTMP协议中专门为 Invoke 开辟了一条通道，在配合 AMF 格式极大的方便了 Flash 下远程服务访问。不过这些都是 Flash下的东西，即使是拥有 Red5 这样的神器让我们在 java 下可以使用 rtmp 但是究其目的还是为了和 flash 通信。一般 flash 调用业务系统的方式还都停留在 http 请求或者通过 red5 服务器代为转发。

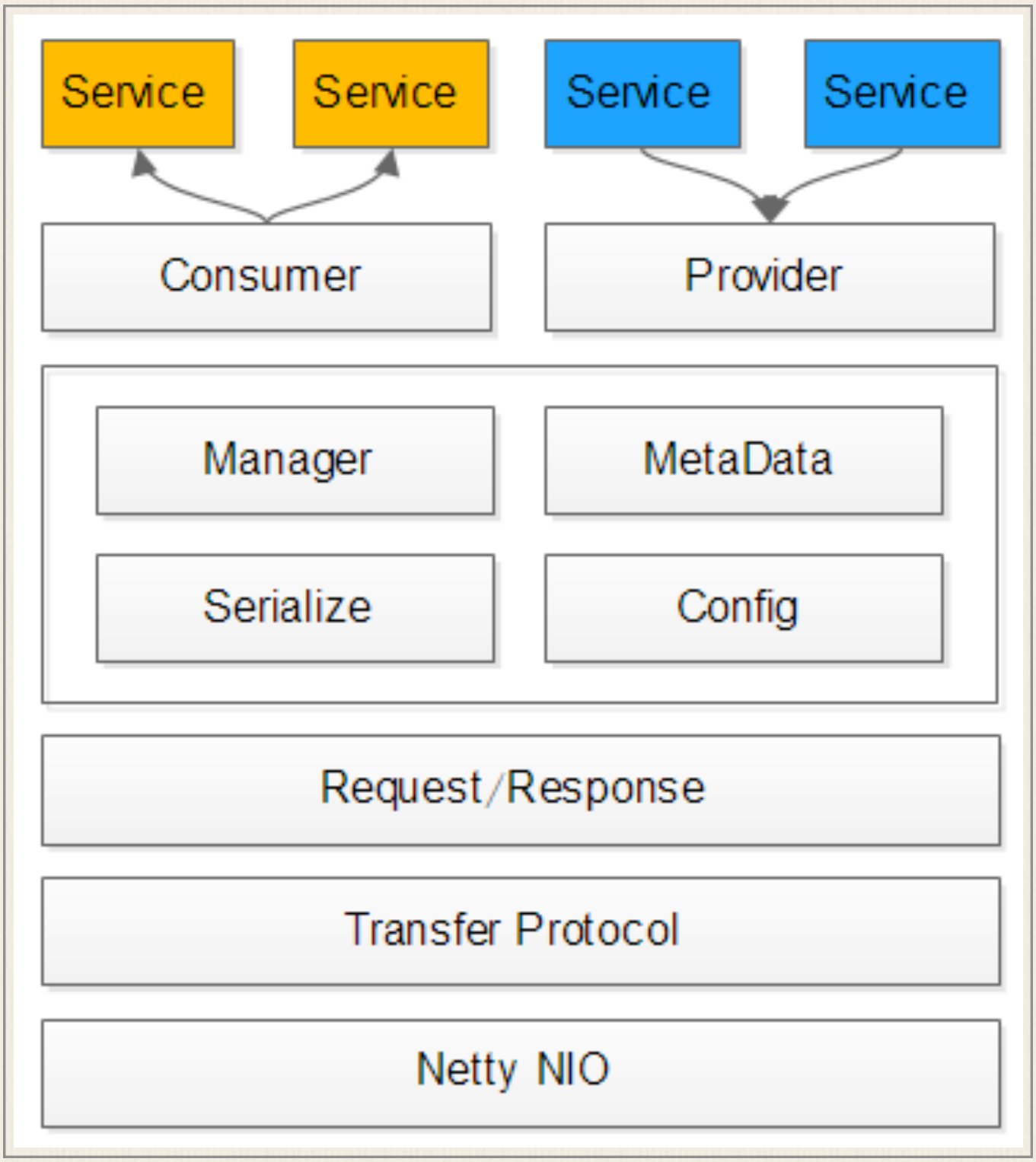
HSF，这个东西是淘宝内部用的很广泛的远程服务框架。它是使用 NIO、Mina 并且工作在长连接模式下。话说这个东西的确是个好东西，淘宝也将其开源了！只可惜，开源了 hsf 但是相关配套依赖没有开源。在加上 hsf 依赖繁杂。这个东西也就只能让局外人膜拜一下，在淘系之外的同学们是无福享受了。

Dubbo，也是淘系的另外一个服务框架，它比较 HSF 来说要轻巧很多。依赖会少一些，这个东东目前也是开源状态。由于我对 dubbo 一点都不了解，在这里保持沉默不做评价。

最后补充一下，真正原生就支持分布式服务调用的也就只有“HSF、Dubbo”至于京东内部是否有更好的解决方案我并不知道。哦还有一点，如果您想脱离 Spring 的话 HSF、Dubbo 会让你失望的。这就是说您的技术构架如果是非 Spring 阵营的会比较悲催。

so，上面提到了很多可用的技术方案，想必最后符合要求也就只有其中 HSF 和 Dubbo 了。为什么其它的方案都不入选呢？原因就是它们虽然可以完成 RPC 但是并不支持分布式。当然您可以通过架设集群来提高它们的可靠性，这些都是您需要额外付出的。

下面这个是 RSF 的架构图，包括服务生产着和消费者在内 RSF 被分为 6 层（网络层、协议层、请求响应层、调度层、接口层、消费者生产者）。



关键5层：

Netty，其中位于最下层的网通信部分 RSF 采用 Netty 实现。Netty 是一款非常优秀的网络通信框架，使用 Netty 可以帮助 RSF 减少大量底层网络上的代码开发。这也就意味着 RSF 将采用 Selector 方式实现异步IO。

Protocol，协议层。该层主要的目的是负责解释翻译 RSF 数据包，并将 RSF 数据包转意成为 Request 和 Response 对象。协议层可以是一个协议栈，这就意味您可以通过 RTMP 、或者其它自定义网络协议传输 RSF 数据包。

Request/Response层，请求响应层。这个在这个层中，RSF 脱离了底层网络方面的特性将每次调用请求对象化为一个 Request 对象，并且将调用结果封装成为一个 Response 对象。这种编程模式和 Web 很像。

调度层，这一层最为复杂。它负责管理本地 RSF 服务的注册，远程传输对象序列化方式的管理，并且还要负责实现其它更加复杂的功能。

接口层，这一层是最终 RSF 暴露给业务系统的接口，将会由两个类提供。一个代表服务生产着，另一个是服务消费者。

序列化格式:

RSF 规定在网络中传输的数据格式可以是任意的。这就意味着您可以使用 AMF 作为 RSF 数据传输格式发布（同时如果协议层支持 RTMP 那您可以在 Flash 中无需通过 red5 这样的中间代理直接访问 RSF 服务）。同样的，如果您使用 Hessian 作为数据传输格式，在其它平台。例如 .net、php。也会很方便的调用 RSF 服务（需要解析 RSF 数据包）。如果协议采用 HTTP，RSF序列化格式采用 JSON ，那么运行在浏览器中的 javascript 也可以绕过 web 服务器，直接访问 RSF 服务。

服务配置Config:

说是服务配置，其实就是路由的功能。先假设我们有4台服务器，其中有两台是位于北京机房，另外两台分别位于青岛和内蒙古。这四台机器上都运行着 RSF，跑着相同的业务系统，这种架构通常前端会有一个 CDN 之类的东西负责让用户就近访问网站。

如果没有服务路由的情况下，用户A在北京即使访问了最近的北京服务器，但是由于调用的 RDS 服务是青岛的，那么也会降低访问速度。因此服务配置所负责的 路由特性可以很方便的高速服务调用程序，优先选用北京机房的 RSF 服务。只有当北京机房的服务撑不住的情况下才会动用其它地域的 RSF 服务。

流量管控：高 级一点的特性是可以通过服务路由来控制服务流量。假如目前要做一个全国范围的活动，我们充分的为每个地方准备了若干机器。但是在活动现场很可能某一个地区 的服务使用量达到了临界点，服务路由应该可以通过配置的方式让附近地区的机器提供一定的流量来减缓这个地区的访问压力。

原文链接：http://my.oschina.net/u/1166271/blog/316686

Docker的生态系统和未来

作者：喻勇

再次纠正概念：Docker不是轻量级容器。它由管理轻量级容器的引擎、客户端和AUFS文件系统三部分组成。轻量级容器（Lightweight Container）在UNIX/Linux 领域经历了十多年的发展，并在最近5年突飞猛进。

轻量级容器技术发展历程

在分析Docker 的生态系统之前，我们首先回顾轻量级容器技术的发展路线图。

- 2000年，BSD Jail：Jail以多种方式改进和增强了BSD类操作系统中用于进程隔离的chroot环境。Jail不仅对文件系统访问实现隔离，还把用户、BSD的网络子系统及一些其他系统资源在内核中进行隔离。
- 2005年，Solaris Containers：Solaris Containers的实现包括两部分：Solaris Zones和System Resource Controls。Zones实现了命名空间隔离和安全访问控制。在non-global zone中的进程既不能看见该zone之外的进程，也不能与之通信。System Resource Controls实现了资源管理功能。
- 2005年，OpenVZ：OpenVZ是GPLv2协议下的开源软件，是基于Linux平台的操作系统级服务器资源隔离解决方案。OpenVZ采用SWsoft的Virutozzo软件产品的内核，Virutozzo是SWsoft公司提供的商业解决方案。

以上这些都是容器技术的先驱，Container 真正开始普及，由以下几个标志性事件推动。

- 2007年，从2.6.4版本开始，cgroups正式进入Linux内核。
- 2008年，LXC 0.10出现，简化了容器的创建和管理。
- 2011年，Linux开发者就容器技术的统一规范达成共识。
- 2012年，Cloud Foundry选择使用WARDEN Container来承载PaaS应用。
- 2013年，Google发布开源容器管理工具lmctfy （Let Me Contain That For You）。

2013年是容器和周边技术高歌猛进的一年，这其中以Docker的流行为代表，以下两家公司和他们的产品具有标志意义。

- 2013年，Docker version 0.10：Docker 是PaaS提供商dotCloud（最近已经正式改名为Docker Inc.）开源的一个基于LXC的高级容器引擎，源代码托管在GitHub上，基于Go语言并遵从Apache 2.0协议开源。Docker的出现极大简化了容器的创建和管理，分层式的AUFS实现了Docker镜像。
- 2013年，CoreOS：这家在硅谷某个车库里成立的创业公司发布了专门为大规模服务器部署定制的Linux精简系统，目的是为运行以轻量级容器为载体的应用提供一个高度优化的底层系统。

2014年，大量围绕Docker和CoreOS的创业公司、新近开源的软件项目、大型企业和互联网公司的加入，使轻量级容器技术的浪潮更上一层楼。

正如定义所言，Docker是“Container Engine”，它是一个把cgroup、namespace等容器底层技术抽象的一个引擎，为用户提供了创建和管理容器的便捷界面（包括命令行和API）。

概念明晰了，我们先从技术栈的维度来看Docker和它的生态系统，把从Linux到Docker做四个层面的分层。

- Linux操作系统。完整的Linux内核，履行操作的使命：管理硬件，调度任务，提供用户界面和服务等。

- 容器的内核实现。这主要包括Linux内核中的cgroup、namespace等，它们为容器（用户进程）的资源隔离性提供了内核层面的保障。

- 轻量级容器的基础工具。通过LXC这样的工具可以完成容器创建、启动等基本操作，但使用LXC需要熟知容器内核实现原理。这对于普通用户来说有一定难度，并且LXC在不同Linux发行版不一致。

- 容器管理引擎。 Docker是这一层的主角。Docker由Docker engine和Docker client组成。Docker engine将神秘的cgroup、复杂的LXC统统隐藏起来，使用简单的命令即可完成容器的运行和管理。它的另一个独特之处在于AUFS的运 用，Copy on write模式的分层文件系统使容器的镜像可以像搭积木一样灵活创建和修改，并在网络上实现增量分发。Docker client，特别是它的API，为在Docker之上的生态系统发展提供了可能性。

Docker的出现和标准化，为以轻量级容器为核心的生态系统提供了爆发式增长的机会。我们从以下几个角度来看Docker的生态系统。

Docker和容器宿主

前文提到的Docker Inc.和CoreOS已经赚足眼球，投资者接踵而至，大规模融资此起彼伏。企业级厂商如红帽、Ubuntu等不甘寂寞，纷纷亮明旗帜，选择站队。

6 月在旧金山举行的DockerCon 2014展示了Docker 对未来的雄心壮志。在Docker engine逐渐稳定并标准化的背景下，Docker的未来目标是为互联网基础架构制定新的标准。最近开源的libcontainer、libchan和 libswarm 三个项目，吹响了这场战役的冲锋号。

- 在新版本Docker engine中，由Go语言开发的libcontainer库已取代LXC。我认为，它更大的目的是反向定义容器的实现标准，将底层实现（也许可以完全不用cgroup甚至Linux）都抽象化到libcontainer的接口。

- libchan类库，以标准接口的方式解决容器的互联互通，实现跨平台，能更好支持分布式系统和并发编程。
- libswarm是另一个很简单的类库，但它将实质性地推动互联网应用架构的创新。它抽象了应用部署和集群管理的细节，为应用程序赋予了跨云平台和互联网级弹性。

CoreOS 的口号“A new way to think about servers”，这句话阐明了他们对改造互联网服务器的目标。CoreOS通过最小化的定制版Linux系统为容器运行提供载体。我曾一度认为 CoreOS的发展方向是与硬件更紧密结合，推出基于ARM的版本，甚至集成进入服务器固件。

然而CoreOS以实际行动证明了我的判断错误：2014年8月14日，传来了CoreOS收购Quay.IO并推出CoreOS Enterprise Registry服务的新闻。

显然，CoreOS并不满足于服务器层的工作，其目标定位在为企业用户提供完整的容器技术服务栈，提供管理大型容器集群的整体解决方案。在这个类别中生存的是标准定义者，它们是整个Docker生态系统的基础。

镜像存储和容器托管

这包括容器的镜像存储和CaaS（Container as a Service）类的容器运行托管，有代表性的公司是StackDock、Orchard、Tutum、Quay.IO、Baremetal.IO等。

这 几家几乎全都是创业公司，他们围绕轻量级容器的整个生命周期来设计自己的产品，有的聚焦容器镜像描述文件（Dockerfile）向导化生成和构建过程的 优化（如StackDock），有的提供包括SSD在内的高性能托管环境（如StackDock和Tutum），有的在监控和弹性扩展方面做足文章（如Tutum），也有像Baremetal.IO这样针对企业级整体解决方案的公司。

容器的镜像存储和运行托管是Docker生态体系中非常 接近最终用户的一层。这个类别中的公司也许并没有高深莫测的技术，也不是标准的定义者，但通过它们与细分市场中客户的长期沟通合作，积累了大量 Docker商用化的经验和实践。这一层最近有两个并购案例：Docker收购Orchard/Fig团队

和CoreOS收购Quay.IO。是不是有点 像大鱼吃小鱼？我们来仔细看看这两家被“吃掉”的公司。

Orchard Laboratories（好邪乎的名字，其实只有两名员工）开发并维护一个名为Fig的开源工具。Fig被称为“by far the easiest way to orchestrate the deployment of multi-container applications”，也被冠以“the perfect Docker companion for developers”。简单地说，Fig以Docker为基础，用容器贯穿整个软件开发流程，快速实现隔离开发环境。Fig让用户编写一个简单的 fig.yml文件列出应用需要的所有Docker容器，以及它们是如何连接在一起的。编写好fig.yml以后，只需要加上-d参数，应用就能开始上线 运行了。这意味着从此开发、测试、运行环境得到统一，容器成为软件发布的新载体。前文提到过Docker的目标是为互联网基础架构制定新的标准，Fig的加入使面向开发者和开发流程这个环节得到极大增强。

Quay.IO，这个团队为用户提供私有的Docker镜像仓库（Image Repository）托管服务。通过这次收购，CoreOS增强了CoreOS Enterprise Registry服务。Quay.IO也只有两名员工。

8月20日，传来了Tutum.co获得260万美元前期投资的新闻，他们是这个领域的大公司（有七名员工），作为CaaS平台提供商，目前有1500个开发者使用其服务。

基于Docker的微PaaS

镜像存储（静态）和容器托管（动态）都是以容器为单位的。下面我们将要讲述以应用为单位，以容器为底层技术实现的微PaaS。

这 几年随着Microsoft Azure、Cloud Foundry的普及（我有幸分别参与了这两个产品在中国市场的早期推广工作），PaaS的概念已经深入人心。传统意义上PaaS实例一般都与一个特定的 IaaS平台绑定，提供部署接口、负载平衡、服务绑定等，然而Docker世界中产生的微PaaS，在此基础上进一步创新。这个领域比较有代表性的是 Flynn和Deis.IO，它们都是开源项目。

Flynn分为Layer 0和Layer 1两层。Layer 0主要做底层硬件和云平台的抽象，分布式配置、任务调度、服务发现等基础工作，它为上层的容器运行环境提供了一个抽象的资源平台。Flynn可以快速部署 在AWS上，今后也可扩

展到其他公有云和私有云。Layer 1主要服务于应用，是PaaS功能的具体实现层，它提供了基本的管理API和客户端，实现了Git Receiver、Heroku Buildpacks、Routing和Datastore等PaaS核心功能。Layer 1本身和它所管理的应用，都以容器为载体。

Deis.IO， 它的一个亮点是用CoreOS承担底层资源管理的任务。在部署Deis PaaS环境时，首先安装的Controller会创建一个CoreOS系统，然后在其之上以容器的方式运行Deis的所有组件。对CoreOS的支持是 一个非常聪明的选择，目前CoreOS已可以运行在多个公有云平台、虚拟机和物理机环境下，这为Deis提供了与生俱来的跨云平台能力。

Flynn 和Deis 的共同特点，是对复杂和大规模分布式应用的原生支持。Heroku创始人Adam Wiggins曾发布著名的“十二要素应用宣言（The Twelve-Factor App）”，这个宣言定义了以服务方式和通过互联网交付的软件应该遵循的十二个要素。Flynn和Deis都是十二要素的忠实拥护者，它们的微PaaS平 台与Heroku有极好的兼容性。

微PaaS创业公司层出不穷，竞争十分激烈，但也许走到最后的只是少数。在这一轮容器技术热潮中，微PaaS正在影响软件开发和运维流程，改变软件的交付方式，把十二要素类互联网应用架构标准化。

Orchestration、Management和Moni-toring

围 绕Docker API做Web UI的门槛相对较低，受到了大家的追捧，这一类主要有DockerUI、Shipyard、maDocker等。它们无一例外都调用Docker API和其他类库，把对容器的管理和监控呈现在Web页面中，这在某种程度上降低了企业网管对这些新技术的恐惧。

这一领域有三个不得不提的高富帅项目：Google Kubernetes、Cloud Foundry的BOSH和Diego。

Kubernetes是构建在Docker之上的容器集群管理系统，Google在2014年6月将这个项目开源。它可以为用户提供跨平台的处理能力，不但能够在Google的基础架构中运行，同时可以访问其他的云计算服务器，如AWS，甚至是私有云。

这 个系统一经开源，就得到了IBM、红帽、微软、Docker、Mesosphere、CoreOS和SaltStack等厂商的支持：微软将确保

Kubernetes能够在其Azure云中作为基于Linux的虚拟机系统容器并正常运作；红帽则将其引入了自己的云产品；IBM的计划是为 Kubernetes与Docker贡献代码；CoreOS将在其操作系统发行版中为Kubernetes提供支持；SaltStack正努力简化 Kubernetes运行在其他环境下的部署流程；而Mesosphere则打算将这项技术加入到自己的Mesos同名开源项目当中。Google一呼百 应的大将之风展露无遗。

Cloud Foundry 的BOSH是部署和运维工具，它通过类似操作系统驱动程序的CPI（Cloud Provider Interface）来实现对多种异构云平台的支持和抽象，以近乎优雅的方式管理VM模板【注：在Cloud Foundry术语中称为干细胞（stemcell）】、软件发布（release）和部署配置脚本文件。最近BOSH推出了一个试验性质的项目BOSH Release for Docker。

Cloud Foundry在它的DEA（Droplet Execution Agent）中使用基于Warden的容器技术来做PaaS的应用隔离。最新的Diego（Go语言版DEA）项目目标是让Cloud Foundry在跨运行时环境方面更具有扩展性，这些运行时环境就包括Docker，也可能会原生支持Windows Server。

网络层的增强和解决方案

容器之间如何互联互通？Docker引擎中的内联网络能否满足企业级别网络的需求？当容器像今天的虚拟机一样在企业环境大规模部署时，复杂的网络需求如网络配置管理、安全监控、流量QoS、网络隔离等一定会出现。

在虚拟化的世界里，这些需求催生了庞大的网络虚拟化（SDN）产业，在容器的环境中，是否有同样的挑战和机会？在这个领域中，目前受关注较多的是Skydock和VNS3开源项目，但整体上还都处在萌芽起步阶段。

谁是容器技术的最终用户

上面列出了很多公司和产品，谁将是容器技术的最终用户呢？我认为会在以下几个领域取得突破。

互联网公司

互 联网公司的开发运维环境复杂，应用多采用分布式架构，后台使用服务的种类繁多，这些都是Docker最擅长解决的问题。据统计，国内外已有一定数量的互联 网公司将Docker集成到内部的开发测试流程，并以Docker

为载体发布应用。GROUPON曾在社区分享他们使用Docker与Jenkins结合做持续集成的案例，国内例如七牛等新兴互联网公司也开始应用Docker。

传统ISV

在整个SDLC（Systems Development Life Cycle）环节中引入Docker，特别是增强以容器为核心的持续集成和持续交付，最终将容器作为软件向云平台交付的实体。这方面目前并没有产品化的整 体解决方案，国外如Shippable，国内如Coding等创业公司在向这个方向努力。

移动开发

这是软件开发最热门的领域，围绕社交、移动、游戏的MBaaS（Mobile Backend as a Service）已有不少成型的产品。Docker，微PaaS如何与移动应用开发相结合，是另一个值得关注的领域。

除此以外，Docker 生态系统在大数据等领域也发展了若干开源工具和项目，这里不一一赘述。

以上是Docker生态系统的一个快照，这个领域的发展可谓一日千里，标准化、开源开放、创业公司、大企业支持、风险投资等特征形成了一个滚雪球的模式，将助推这一轮技术革新到更高的一个台阶。

Docker的未来

接下来Docker会有哪些新的进展？它到底是极客手里昙花一现的技术玩具，还是下一代的互联网基础架构?

Docker 创始人Solomon Hykes在DockerCon 2014上放出了“Upgrade the Internet”的豪言壮语。目前的Docker和它周边的生态系统，距离完成这个伟大使命，还有多远呢？行文至此，我尝试结合自己推广和建设 Cloud Foundry生态系统的一些经验，对此做初步分析，抛砖引玉，意在引起更多针对轻量级容器技术的深度思考。

首先需要思考，Internet为什么需要Upgrade?

近些年云计算高速发展，日趋成熟的IaaS平台，解决了以自动化方式组织、管理和使用大规模硬件资源方面的需求（图1），但应用架构层面的演进不容乐观。

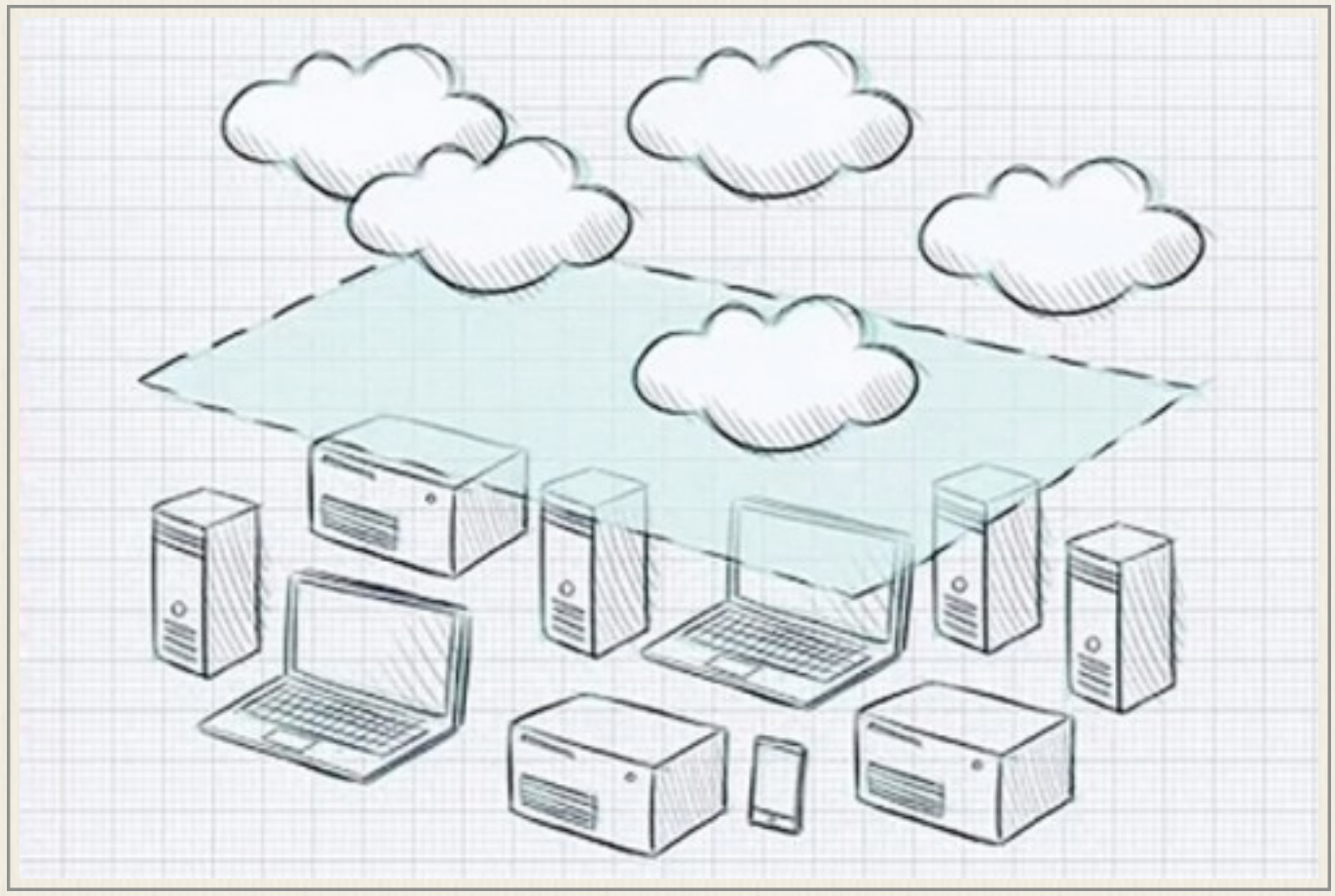


图1 云计算的自动化组织

传统的三层架构应用，往往通过虚拟机的方式在云平台部署，并未针对云计算平台的特点做充分的优化；复杂的分布式互联网应用，多数通过与底层特定云平 台紧密绑定的DevOps工具来部署和管理，缺少跨云平台的灵活性。试想如果我们把每一种类型的IaaS都看做一类品牌的服务器，我们的应用实际上是与硬 件紧耦合的。那么这跟越过操作系统，直接针对CPU的机器指令开发程序，有何区别呢?

Docker的出现，不仅为使用Linux轻量级容器提供了便利的工具，更重要的是它将引发互联网应用架构的革命。主要体现在以下几个方面。

1. 以容器为开发、测试和发布的单元，将使单机、私有云、公有云的界限模糊，让开发者更加关注应用开发本身，显著降低DevOps的压力（缺乏复杂分布式互联网应用的运维能力，是阻碍传统企业转型互联网架构的门槛之一）。

2. 传 统应用在云平台上，仍旧面临高可用性，数据吞吐瓶颈和安全的考验，特别是大容量和大流量的数据库节点，是企业应用在互联网架构下

获得弹性的一大障碍。数据 和服务，是否可以做到分布式，这是目前架构师面临的巨大考验。轻量级容器在快速启动、一致性、服务托管、微服务等方面的能力，是否能为突破这层障碍提供可 能?

3. Docker的热潮推动了标准的形成，DockerCon 2014上公布的三个类库：libcontainer、libchan、libswarm是这一场革命的三大基石。以此为规范构建的互联网应用，是否能获得跨互联网的弹性和可用性?

罗马并非一日之间建成，Docker、轻量级容器以及互联网应用架构的革命还有很长的路要走。我认为，以下几个领域将会直接影响Docker未来的发展。

开源的独立性

有 关Docker的生态系统，前文已经有详细分析，这里不再赘述。这么短时间内，便相继传来了Docker获得追加投资，Fig、Quay.IO等被收购的 新闻，相信在这个领域，融资、并购的好戏还将愈演愈烈。资本市场的参与固然会加速技术的发展，但投资者的中短期获利预期，势必引起未来12~18个月内更 多的再融资、并购甚至套现。这类资本活动，以及一些大型商业公司的参与，是否会过早带来商业性的盈利诉求，影响开源的独立性和中立发展，是一个需要特别关 注的问题。

如何被企业用户接受

Docker如何走进企业级市场? 除了技术上需要在安全、隔离性等方面提供更多增强，选择合适的切入点至关重要。从需求来说，企业级客户面临如下几个挑战。

- 在传统企业纷纷触网，业务向互联网转型的大背景下，原有的企业应用和支持平台，能否承担互联网带来的爆发式流量、压力和弹性?
- 企业IT运维人员，是否具备DevOps的能力?
- 从私有云到公有云的过渡，是否能以更低的成本完成?

我认为，Docker对ISV厂商来说是一次重要的机会，它不仅有助于统一开发流程中各类异构开发环境，降低开发测试的成本，更重要的是以容器方式交付的软 件，能为客户带来切实的价值。Docker与生俱来的可移植性、日渐成为标准的底层平台接口、跨云的API等，能在一定程度上解决企业客户上述面临的烦 恼，为企业级客户带来战略性的价值。

Docker对私有云运营商也是一次机会，提供满足企业级需求的容器托管平台，不论是CaaS，还是微PaaS，如果达到企业级用户对性能、安全、监控、管理、合规等方面的要求，都将获得市场的认可。当软件的容器式交付成为标准，这也许会带来私有云市场的一次新的洗牌。

对上层应用架构的影响

所有的云应用架构师都应该熟读著名的“十二要素应用宣言”。Docker和它周边的系统，是否能成为运行十二要素类应用的原生平台？是否能为这类应用带来跨云和分布式数据服务的能力?

十二要素强调不会区别对待本地或第三方服务，应用把不同来源的服务都当作资源，并与之保持松耦合绑定。Dokku的作者以及Docker早期的贡献者Jeff Lindsay在CenturyLink的一个采访中讨论了如何解决涉及面向Docker服务的架构的问题。

社 区目前有很多努力集中在：服务发现系统（Etcd、Discovered等），远程服务代理（Ambassadord），服务注册 （Registrator），分布式服务发现和路由网格，libchan之上的通信协议，无中心化的软件配置文件，分布式调度系统等，这些领域的创新都处 在起步阶段。6~12个月内这方面的进展将在PaaS和软件架构领域带来新的变化，需要紧密关注。

Docker是一个不超过18个月的新生儿，它背后的技术也并非高深莫测，备受关注的原因主要是适应互联网时代的需要，为目前的云平台和应用架构指明了一个新的发展方向。让我们拭目以待，关注并参与这一场即将到来的“Generational Shift”。

原文链接：http://www.csdn.net/article/2014-09-24/2821832

Docker源码分析（一）：Docker架构

作者：孙宏亮

1 背景

1.1 Docker简介

Docker是Docker公司开源的一个基于轻量级虚拟化技术的容器引擎项目，整个项目基于Go语言开发，并遵从Apache 2.0协议。目前，Docker可以在容器内部快速自动化部署应用，并可以通过内核虚拟化技术（namespaces及cgroups等）来提供容器的资 源隔离与安全保障等。由于Docker通过操作系统层的虚拟化实现隔离，所以Docker容器在运行时，不需要类似虚拟机（VM）额外的操作系统开销，提 高资源利用率，并且提升诸如IO等方面的性能。

由于众多新颖的特性以及项目本身的开放性，Docker在不到两年的时间里迅速获得诸多厂商的青睐，其中更是包括Google、Microsoft、VMware等业界行业领导者。Google在今年六月份推出了Kubernetes，提供Docker容器的调度服务，而今年8月Microsoft宣布Azure上支持Kubernetes，随后传统虚拟化巨头VMware宣布与Docker强强合作。今年9月中旬，Docker更是获得4000万美元的C轮融资，以推动分布式应用方面的发展。

从目前的形势来看，Docker的前景一片大好。本系列文章从源码的角度出发，详细介绍Docker的架构、Docker的运行以及Docker的卓越特性。本文是Docker源码分析系列的第一篇——Docker架构篇。

1.2 Docker版本信息

本文关于Docker架构的分析都是基于Docker的源码与Docker相应版本的运行结果，其中Docker为最新的1.2版本。

2 Docker架构分析内容安排

本文的目的是：在理解Docker源代码的基础上，分析Docker架构。分析过程中主要按照以下三个步骤进行：

- Docker的总架构图展示
- Docker架构图内部各模块功能与实现分析
- 以Docker命令的执行为例，进行Docker运行流程阐述

3 Docker总架构图

学习Docker的源码并不是一个枯燥的过程，反而可以从中理解Docker架构的设计原理。Docker对使用者来讲是一个C/S模式的架构，而Docker的后端是一个非常松耦合的架构，模块各司其职，并有机组合，支撑Docker的运行。

在此，先附上Docker总架构，如图3.1。

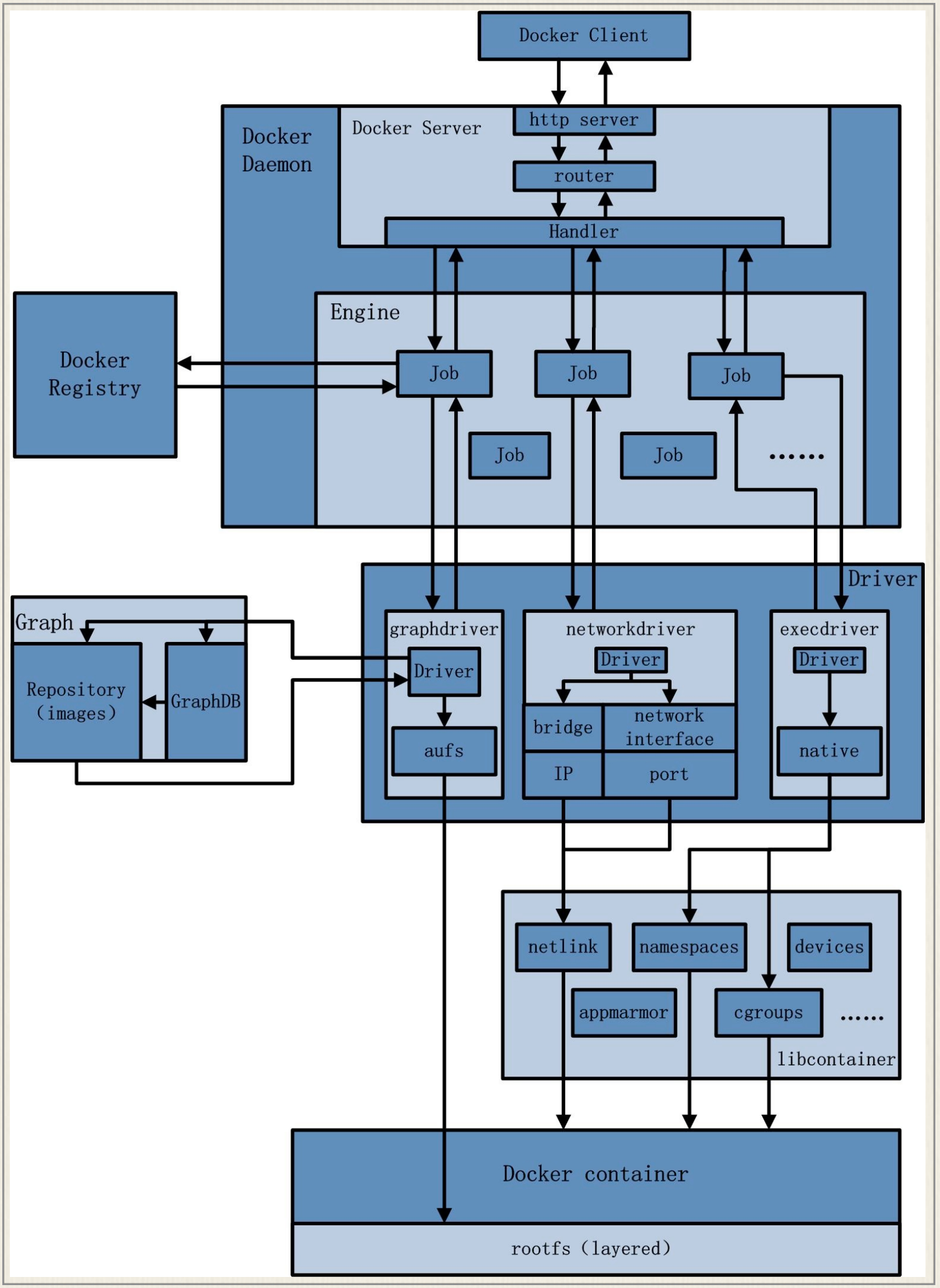


图3.1 Docker总架构图

如图3.1，不难看出，用户是使用Docker Client与Docker Daemon建立通信，并发送请求给后者。

而Docker Daemon作为Docker架构中的主体部分，首先提供Server的功能使其可以接受Docker Client的请求；而后Engine执行Docker内部的一系列工作，每一项工作都是以一个Job的形式的存在。

Job的运行过程中，当需要容器镜像时，则从Docker Registry中下载镜像，并通过镜像管理驱动graphdriver将下载镜像以Graph的形式存储；当需要为Docker创建网络环境时，通过网 络管理驱动networkdriver创建并配置Docker容器网络环境；当需要限制Docker容器运行资源或执行用户指令等操作时，则通过 execdriver来完成。

而libcontainer是一项独立的容器管理包，networkdriver以及execdriver都是通过libcontainer来实现具体对容器进行的操作。

当执行完运行容器的命令后，一个实际的Docker容器就处于运行状态，该容器拥有独立的文件系统，独立并且安全的运行环境等。

4 Docker架构内各模块的功能与实现分析

接下来，我们将从Docker总架构图入手，抽离出架构内各个模块，并对各个模块进行更为细化的架构分析与功能阐述。主要的模块有：Docker Client、Docker Daemon、Docker Registry、Graph、Driver、libcontainer以及Docker container。

4.1 Docker Client

Docker Client是Docker架构中用户用来和Docker Daemon建立通信的客户端。用户使用的可执行文件为docker，通过docker命令行工具可以发起众多管理container的请求。

Docker Client可以通过以下三种方式和Docker Daemon建立通信：tcp://host:port，unix://path_to_socket和fd://socketfd。为了简单起见，本文 一律使用第一种方式作为讲述两者通信的原型。与此同时，与Docker Daemon建立连接并传输请求的时候，Docker Client可以通过设置命令行flag参数的形式设置安全传输层协议(TLS)的有关参数，保证传输的安全性。

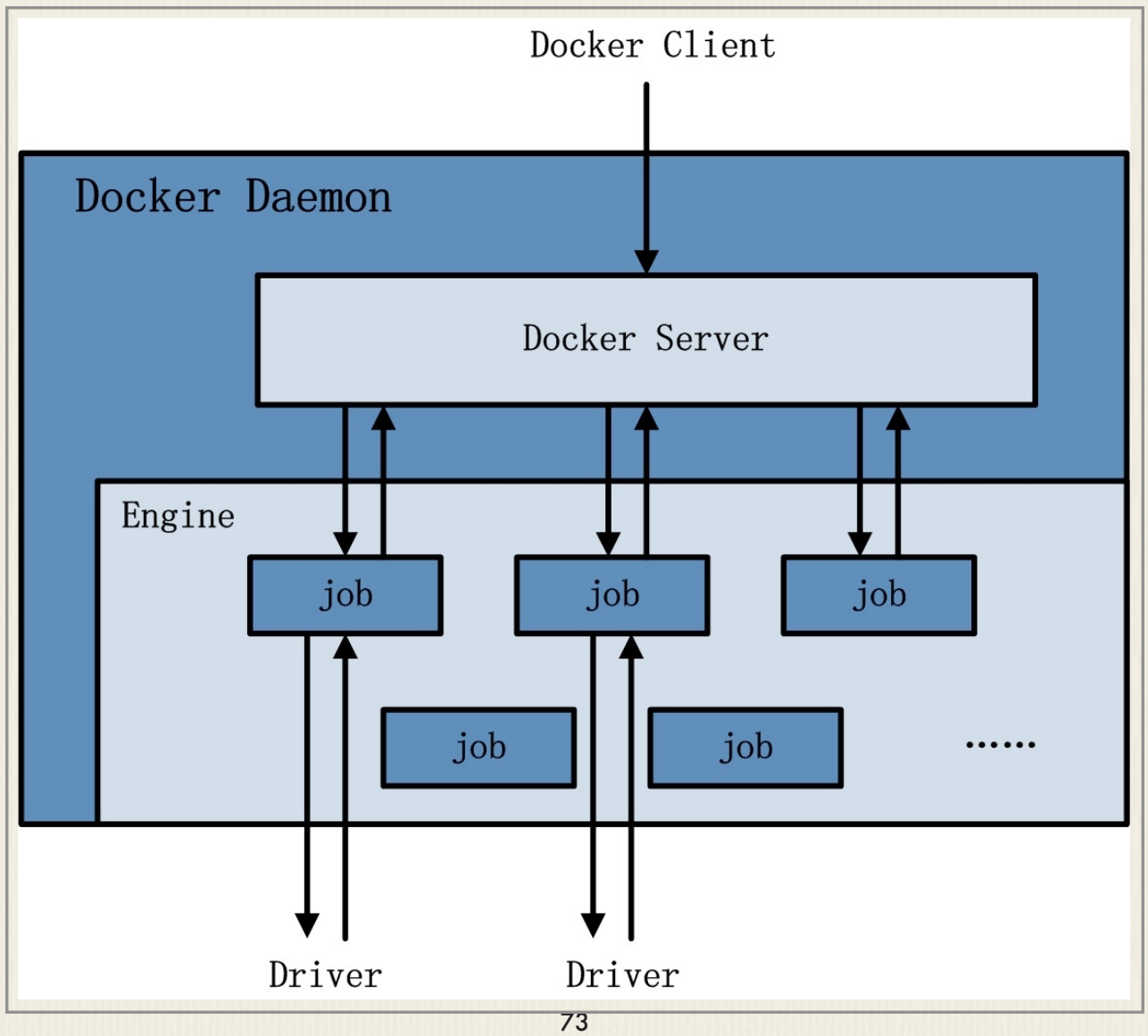
Docker Client发送容器管理请求后，由Docker Daemon接受并处理请求，当Docker Client接收到返回的请求相应并简单处理后，Docker Client一次完整的生命周期就结束了。当需要继续发送容器管理请求时，用户必须再次通过docker可执行文件创建Docker Client。

4.2 Docker Daemon

Docker Daemon是Docker架构中一个常驻在后台的系统进程，功能是：接受并处理Docker Client发送的请求。该守护进程在后台启动了一个Server，Server负责接受Docker Client发送的请求；接受请求后，Server通过路由与分发调度，找到相应的Handler来执行请求。

Docker Daemon启动所使用的可执行文件也为docker，与Docker Client启动所使用的可执行文件docker相同。在docker命令执行时，通过传入的参数来判别Docker Daemon与Docker Client。

Docker Daemon的架构，大致可以分为以下三部分：Docker Server、Engine和Job。Daemon架构如图4.1。



4.2.1 Docker Server

Docker Server在Docker架构中是专门服务于Docker Client的server。该server的功能是：接受并调度分发Docker Client发送的请求。Docker Server的架构如图4.2。

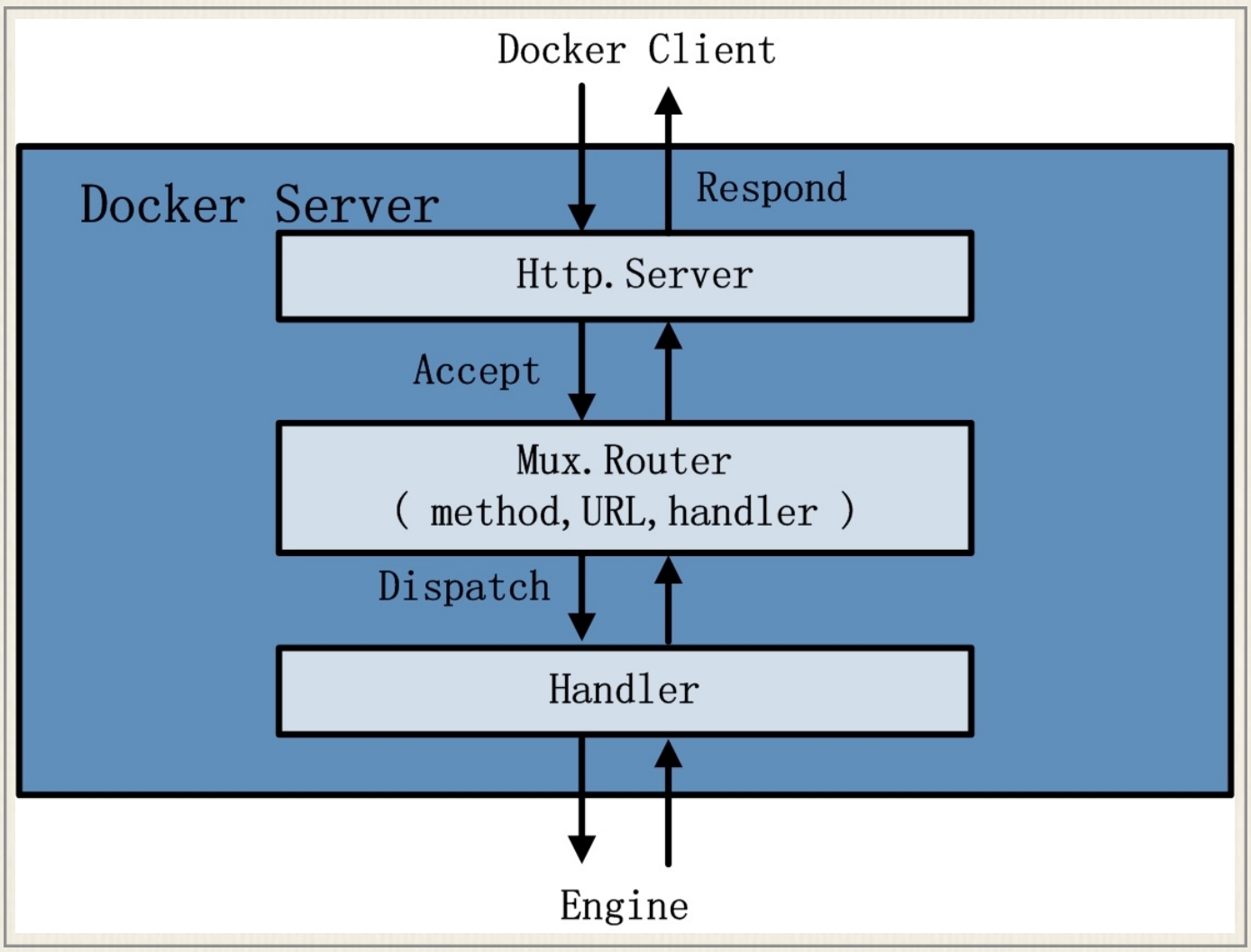


图4.2 Docker Server架构示意图

在Docker的启动过程中，通过包gorilla/mux，创建了一个mux.Router，提供请求的路由功能。在Golang 中，gorilla/mux是一个强大的URL路由器以及调度分发器。该mux.Router中添加了众多的路由项，每一个路由项由HTTP请求方法 （PUT、POST、GET或DELETE）、URL、Handler三部分组成。

若Docker Client通过HTTP的形式访问Docker Daemon，创建完mux.Router之后，Docker将Server的监听地址以及mux.Router作为参数，创建一个 httpSrv=http.Server{}，最终执行httpSrv.Serve()为请求服务。

在Server的服务过程中，Server在listener上接受Docker Client的访问请求，并创建一个全新的goroutine来服务该请求。在goroutine中，首先读取请求内容，然后做解析工作，接着找到相应的 路由项，随后调用相应的Handler来处理该请求，最后Handler处理完请求之后回复该请求。

需要注意的是：Docker Server的运行在Docker的启动过程中，是靠一个名为"serveapi"的job的运行来完成的。原则上，Docker Server的运行是众多job中的一个，但是为了强调Docker Server的重要性以及为后续job服务的重要特性，将该"serveapi"的job单独抽离出来分析，理解为Docker Server。

4.2.2 Engine

Engine是Docker架构中的运行引擎，同时也Docker运行的核心模块。它扮演Docker container-
er存储仓库的角色，并且通过执行job的方式来操纵管理这些容器。

在Engine数据结构的设计与实现过程中，有一个handler对象。该handler对象存储的都是关于众多特定job的handler处理访 问。举例说明，Engine的handler对象中有一项为：{"create": daemon.ContainerCreate,}，则说明当名为"create"的job在运行时，执行的是 daemon.ContainerCreate的handler。

4.2.3 Job

一个Job可以认为是Docker架构中Engine内部最基本的工作执行单元。Docker可以做的每一项工作，都可以抽象为一个job。例如： 在容器内部运行一个进程，这是一个job；创建一个新的容器，这是一个job，从Internet上下载一个文档，这是一个job；包括之前在 Docker Server部分说过的，创建Server服务于HTTP的API，这也是一个job，等等。

Job的设计者，把Job设计得与Unix进程相仿。比如说：Job有一个名称，有参数，有环境变量，有标准的输入输出，有错误处理，有返回状态等。

4.3 Docker Registry

Docker Registry是一个存储容器镜像的仓库。而容器镜像是在容器被创建时，被加载用来初始化容器的文件架构与目录。

在Docker的运行过程中，Docker Daemon会与Docker Registry通信，并实现搜索镜像、下载镜像、上传镜像三个功能，这三个功能对应的job名称分别为"search"，"pull" 与 "push"。

其中，在Docker架构中，Docker可以使用公有的Docker Registry，即大家熟知的Docker Hub，如此一来，Docker获取容器镜像文件时，必须通过互联网访问Docker Hub；同时Docker也允许用户构建本地私有的Docker Registry，这样可以保证容器镜像的获取在内网完成。

4.4 Graph

Graph在Docker架构中扮演已下载容器镜像的保管者，以及已下载容器镜像之间关系的记录者。一方面，Graph存储着本地具有版本信息的文件系统镜像，另一方面也通过GraphDB记录着所有文件系统镜像彼此之间的关系。Graph的架构如图4.3。

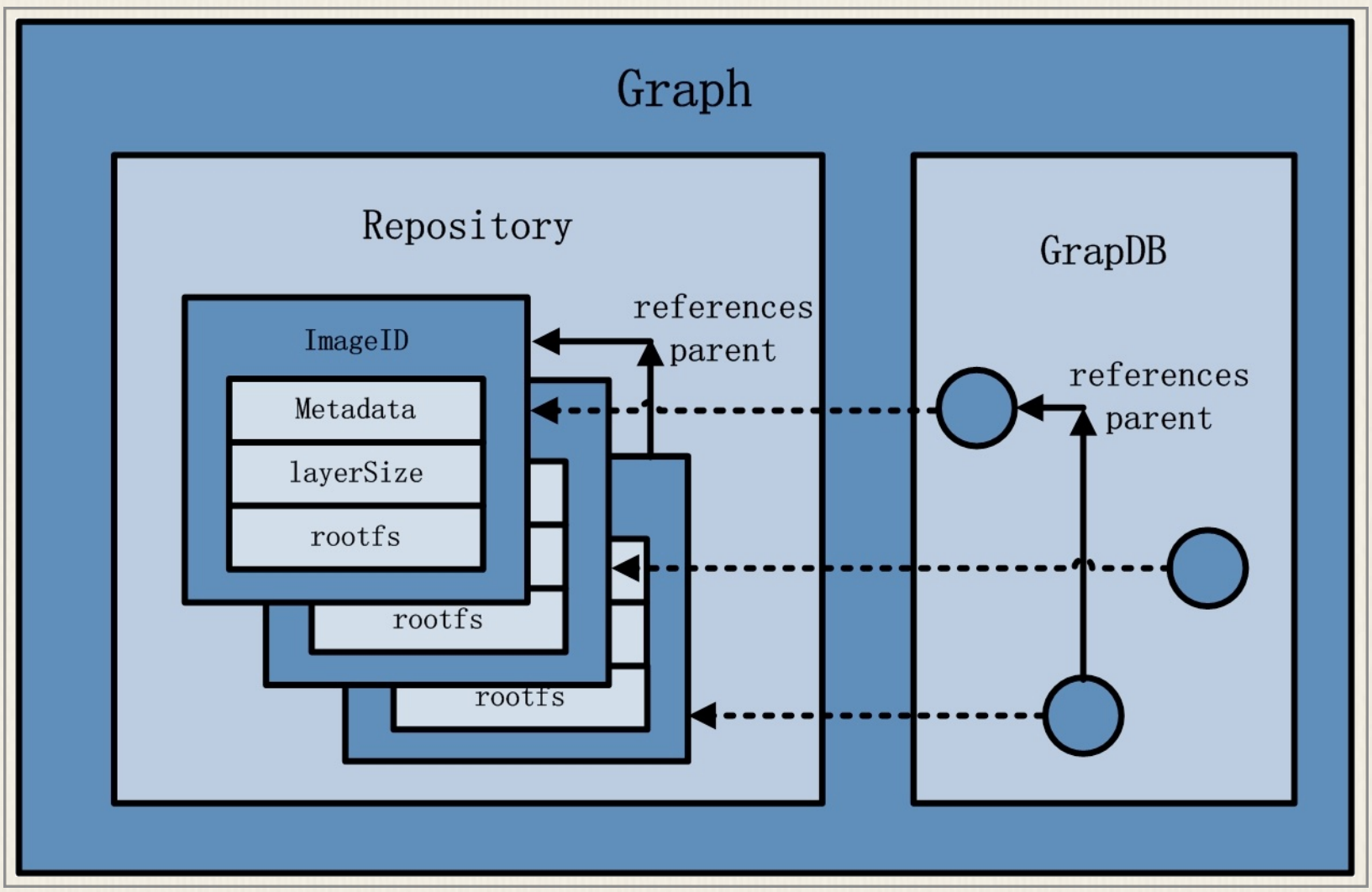


图4.3 Graph架构示意图

其中，GraphDB是一个构建在SQLite之上的小型图数据库，实现了节点的命名以及节点之间关联关系的记录。它仅仅实现了大多数图数据库所拥有的一个小的子集，但是提供了简单的接口表示节点之间的关系。

同时在Graph的本地目录中，关于每一个的容器镜像，具体存储的信息有：该容器镜像的元数据，容器镜像的大小信息，以及该容器镜像所代表的具体rootfs。

4.5 Driver

Driver是Docker架构中的驱动模块。通过Driver驱动，Docker可以实现对Docker容器执行环境的定制。由于Docker运 行的生命周期中，并非用户所有的操作都是针对Docker容器的管理，另外还有关于Docker运行信息的获取，Graph的存储与记录等。因此，为了将 Docker容器的管理从Docker Daemon内部业务逻辑中区分开来，设计了Driver层驱动来接管所有这部分请求。

在Docker Driver的实现中，可以分为以下三类驱动：graphdriver、networkdriver和execdriver。

graphdriver主要用于完成容器镜像的管理，包括存储与获取。即当用户需要下载指定的容器镜像时，graphdriver将容器镜像存储在 本地的指定目录；同时当用户需要使用指定的容器镜像来创建容器的rootfs时，graphdriver从本地镜像存储目录中获取指定的容器镜像。

在graphdriver的初始化过程之前，有4种文件系统或类文件系统在其内部注册，它们分别是aufs、btrfs、vfs和 devmapper。而Docker在初始化之时，通过获取系统环境变量”DOCKER_DRIVER”来提取所使用driver的指定类型。而之后所有 的graph操作，都使用该driver来执行。

graphdriver的架构如图4.4：

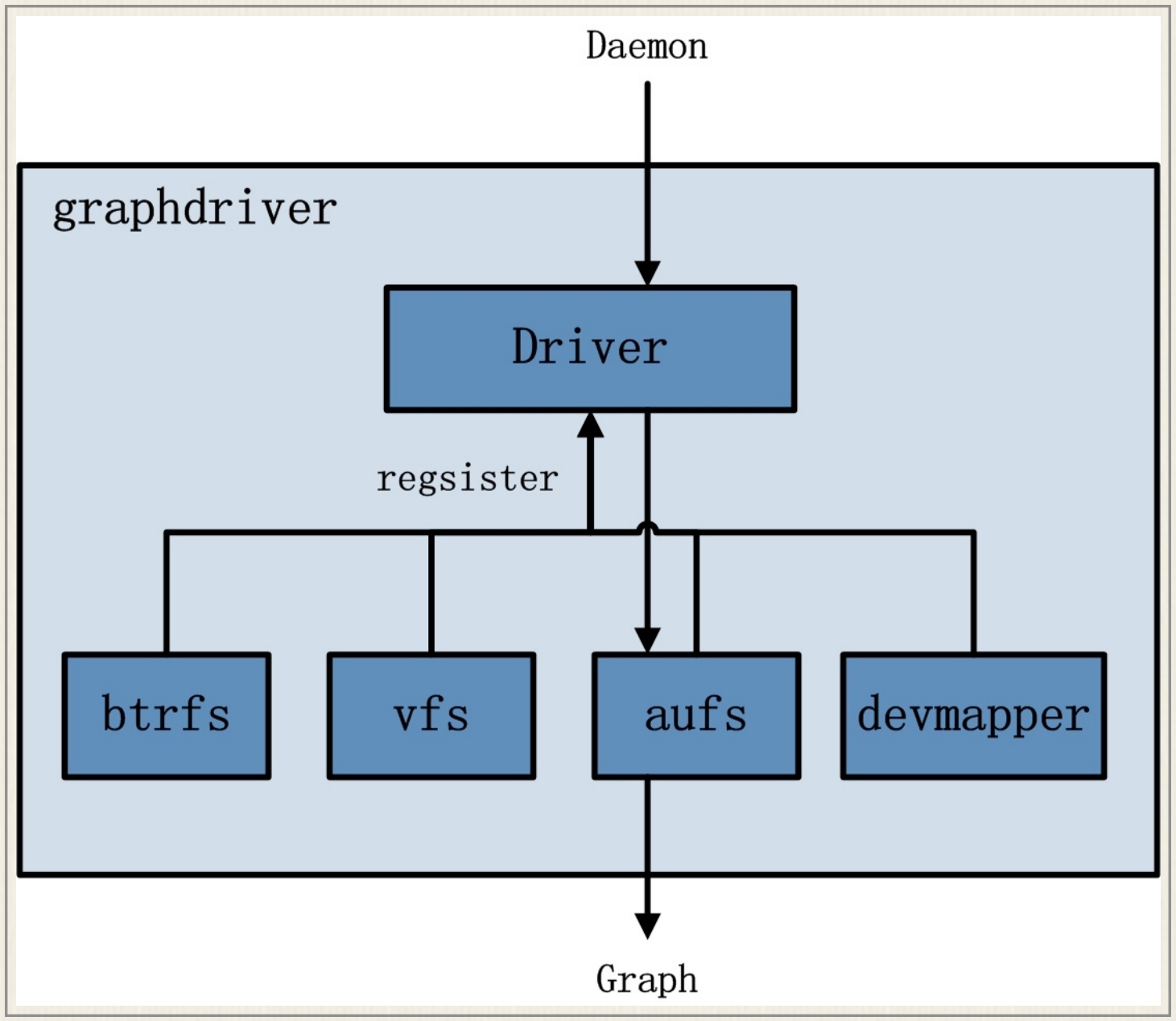


图4.4 graphdriver架构示意图

networkdriver的用途是完成Docker容器网络环境的配置，其中包括Docker启动时为Docker环境创建网桥；Docker容 器创建时为其创建专属虚拟网卡设备；以及为Docker容器分配IP、端口并与宿主机做端口映射，设置容器防火墙策略等。networkdriver的架 构如图4.5：

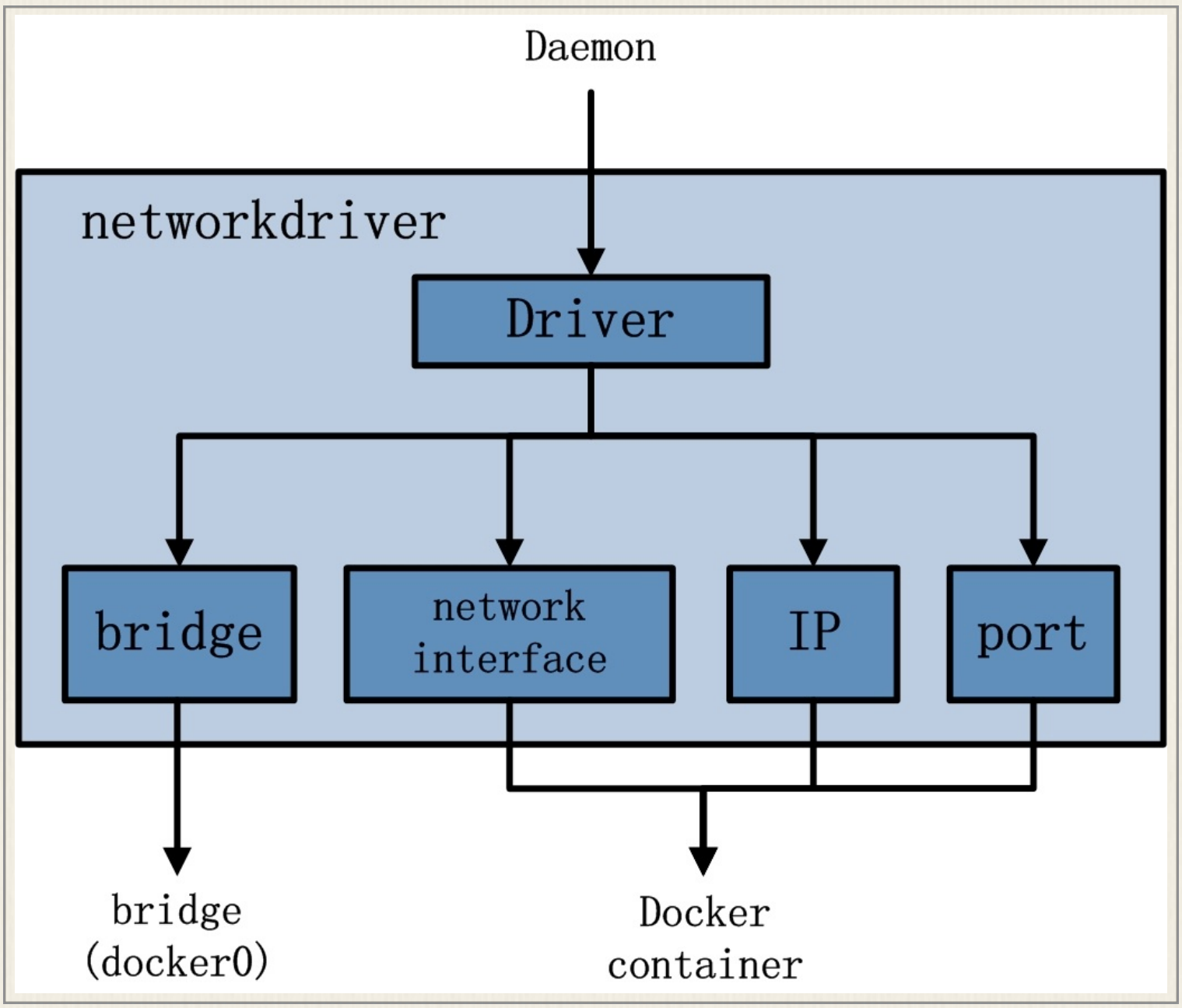


图4. 5 networkdriver架构示意图

execdriver作为Docker容器的执行驱动，负责创建容器运行命名空间，负责容器资源使用的统计与限制，负责容器内部进程的真正运行等。在execdriver的实现过程中，原先可以使用LXC驱动调用LXC的接口，来操纵容器的配置以及生命周期，而现在execdriver默认使用 native驱动，不依赖于LXC。具体体现在Daemon启动过程中加载的ExecDriverflag参数，该参数在配置文件已经被设 为"native"。这可以认为是Docker在1.2版本上一个很大的改变，或者说Docker实现跨平台的一个先兆。execdriver架构如图4.6：

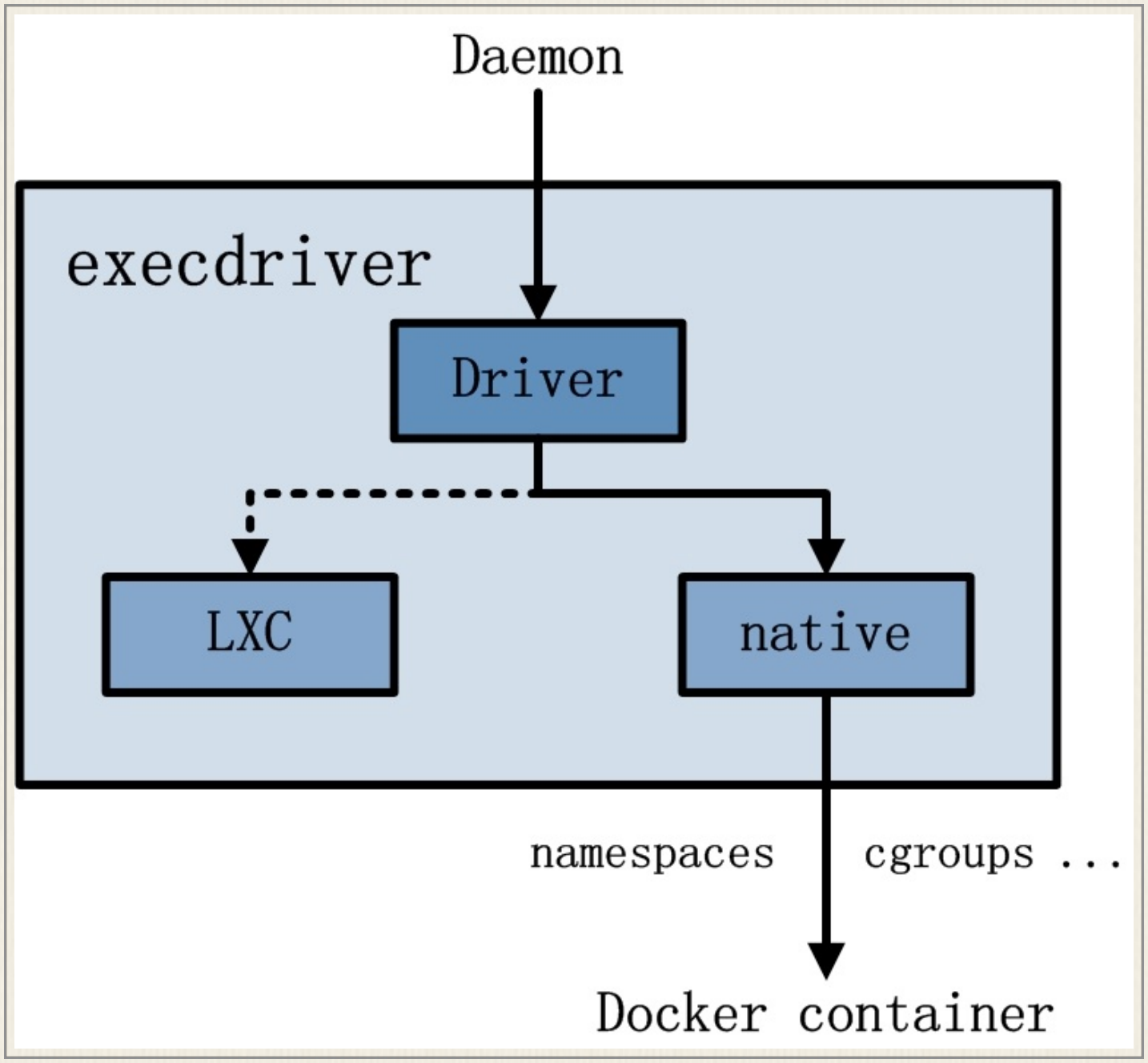


图4.6 execdriver架构示意图

4.6 libcontainer

libcontainer是Docker架构中一个使用Go语言设计实现的库，设计初衷是希望该库可以不依靠任何依赖，直接访问内核中与容器相关的API。

正是由于libcontainer的存在，Docker可以直接调用libcontainer，而最终操纵容器的namespace、 cgroups、apparmor、网络设备以及防火墙规则等。这一系列操作的完成都不需要依赖LXC或者其他包。libcontainer架构如图 4.7：

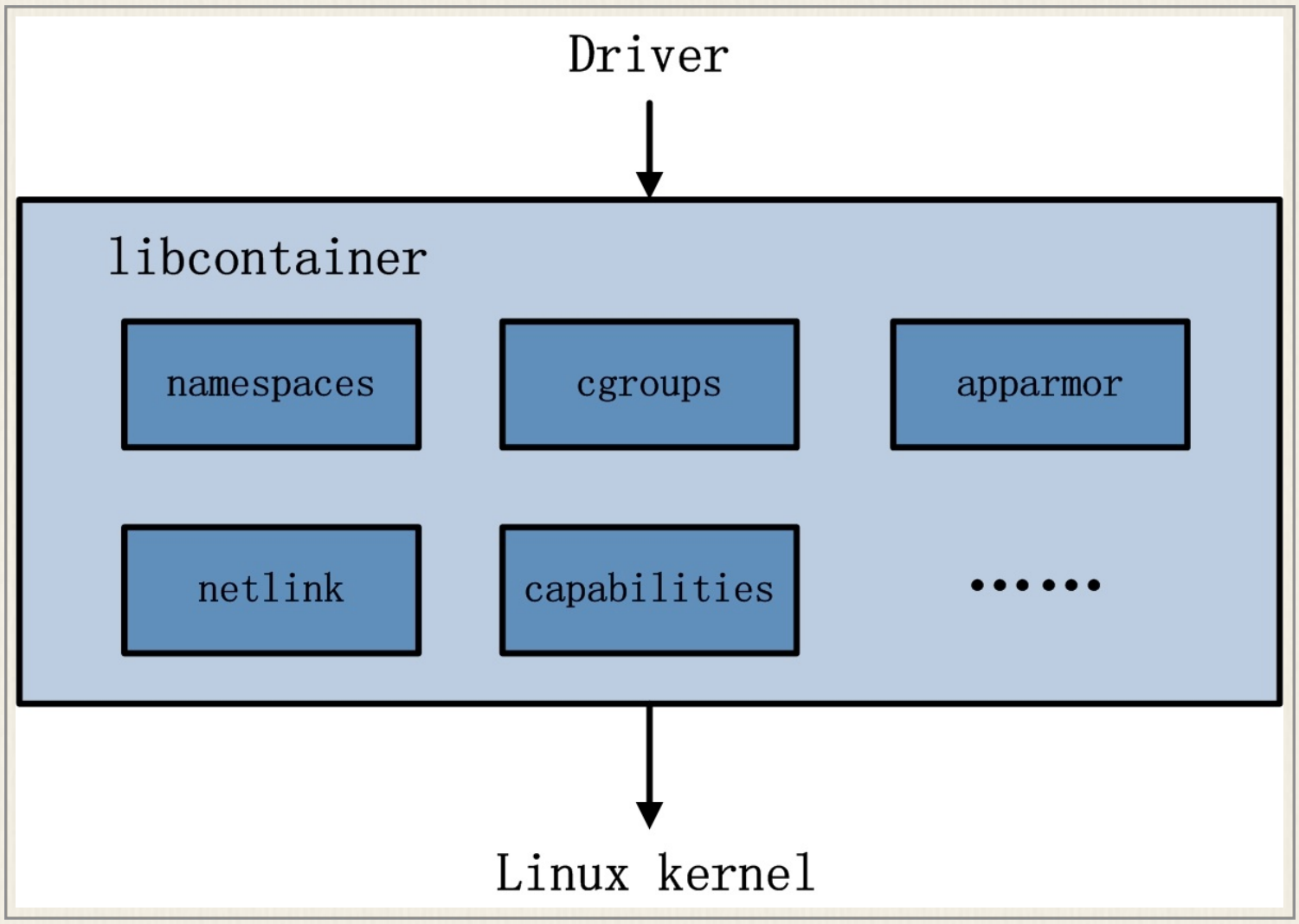


图4.7 libcontainer示意图

另外，libcontainer提供了一整套标准的接口来满足上层对容器管理的需求。或者说，libcontainer屏蔽了Docker上层对容 器的直接管理。又由于libcontainer使用Go这种跨平台的语言开发实现，且本身又可以被上层多种不同的编程语言访问，因此很难说，未来的 Docker就一定会紧紧地和Linux捆绑在一起。而于此同时，Microsoft在其著名云计算平台Azure中，也添加了对Docker的支持，可 见Docker的开放程度与业界的火热度。

暂不谈Docker，由于libcontainer的功能以及其本身与系统的松耦合特性，很有可能会在其他以容器为原型的平台出现，同时也很有可能催生出云计算领域全新的项目。

4.7 Docker container

Docker container（Docker容器）是Docker架构中服务交付的最终体现形式。

Docker按照用户的需求与指令，订制相应的Docker容器：

- 用户通过指定容器镜像，使得Docker容器可以自定义rootfs等文件系统；
- 用户通过指定计算资源的配额，使得Docker容器使用指定的计算资源；
- 用户通过配置网络及其安全策略，使得Docker容器拥有独立且安全的网络环境；
- 用户通过指定运行的命令，使得Docker容器执行指定的工作。

Docker容器示意图如图4.8：

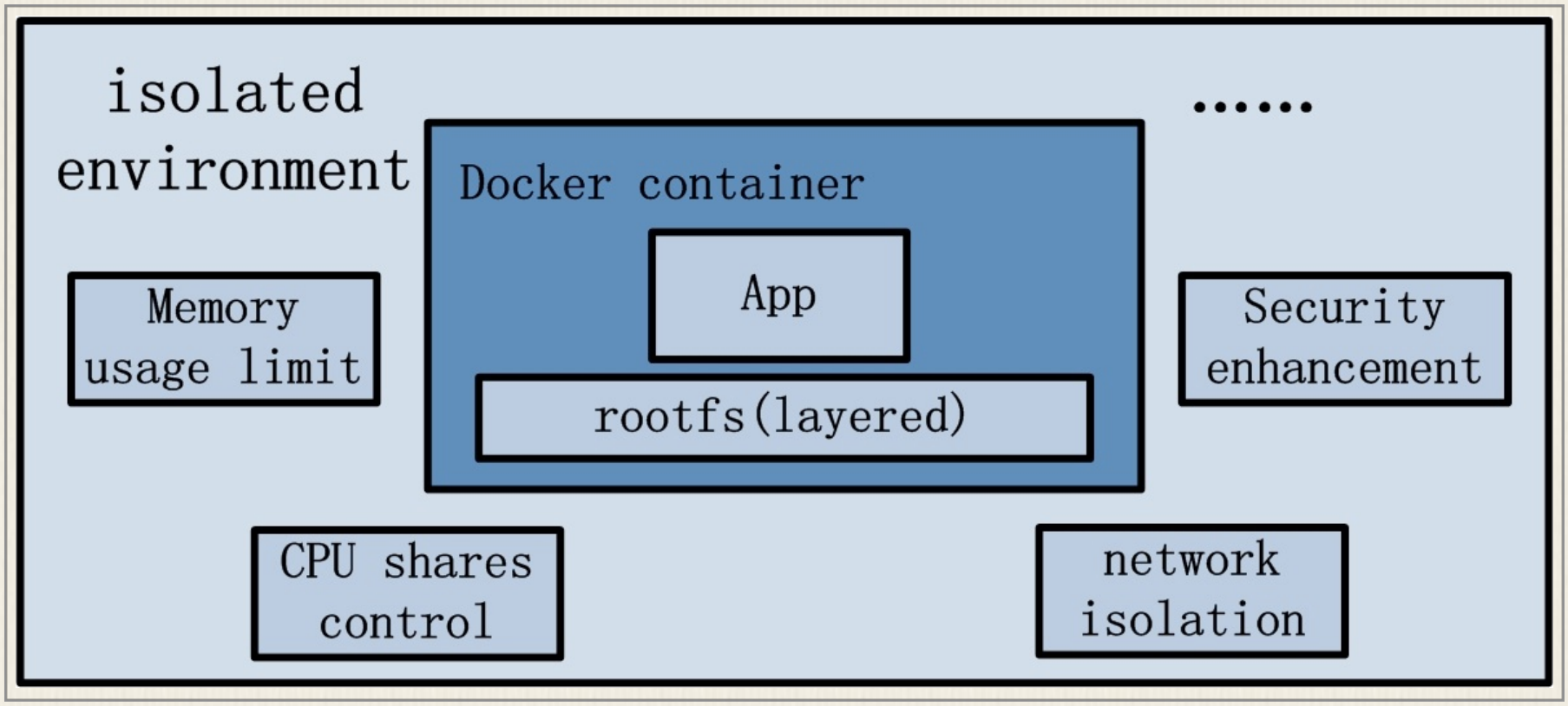


图4.8 Docker容器示意图

5 Docker运行案例分析

上一章节着重于Docker架构中各个部分的介绍。本章的内容，将以串联Docker各模块来简要分析，分析原型为Docker中的docker pull与docker run两个命令。

5.1 docker pull

docker pull命令的作用为：从Docker Registry中下载指定的容器镜像，并存储在本地的Graph中，以备后续创建Docker容器时的使用。docker pull命令执行流程如图5.1。

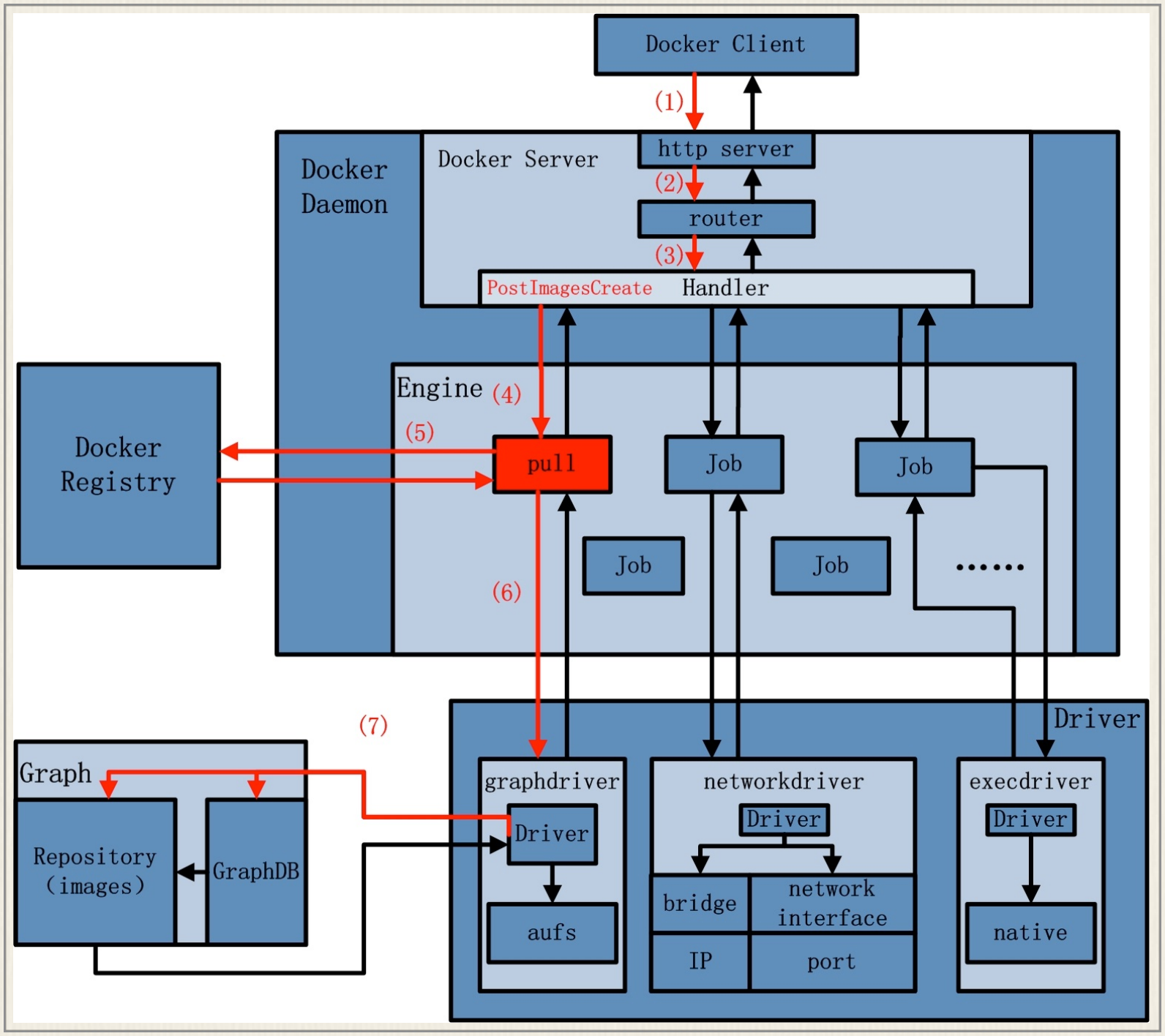


图5.1 docker pull命令执行流程示意图

如图，图中标记的红色箭头表示docker pull命令在发起后，Docker所做的一系列运行。以下逐一分析这些步骤。

(1) Docker Client接受docker pull命令，解析完请求以及收集完请求参数之后，发送一个HTTP请求给Docker Server，HTTP请求方法为POST，请求URL为"/images/create? "+"xxx"；

(2) Docker Server接受以上HTTP请求，并交给mux.Router，mux.Router通过URL以及请求方法来确定执行该请求的具体handler；

(3) mux.Router将请求路由分发至相应的handler，具体为PostImagesCreate；

(4) 在PostImageCreate这个handler之中，一个名为"pull"的job被创建，并开始执行；

(5) 名为"pull"的job在执行过程中，执行pullRepository操作，即从Docker Registry中下载相应的一个或者多个image；

(6) 名为"pull"的job将下载的image交给graphdriver；

(7) graphdriver负责将image进行存储，一方创建graph对象，另一方面在GraphDB中记录image之间的关系。

5.2 docker run

docker run命令的作用是在一个全新的Docker容器内部运行一条指令。Docker在执行这条命令的时候，所做工作可以分为两部分：第一，创建Docker 容器所需的rootfs；第二，创建容器的网络等运行环境，并真正运行用户指令。因此，在整个执行流程中，Docker Client给Docker Server发送了两次HTTP请求，第二次请求的发起取决于第一次请求的返回状态。Docker run命令执行流程如图5.2。

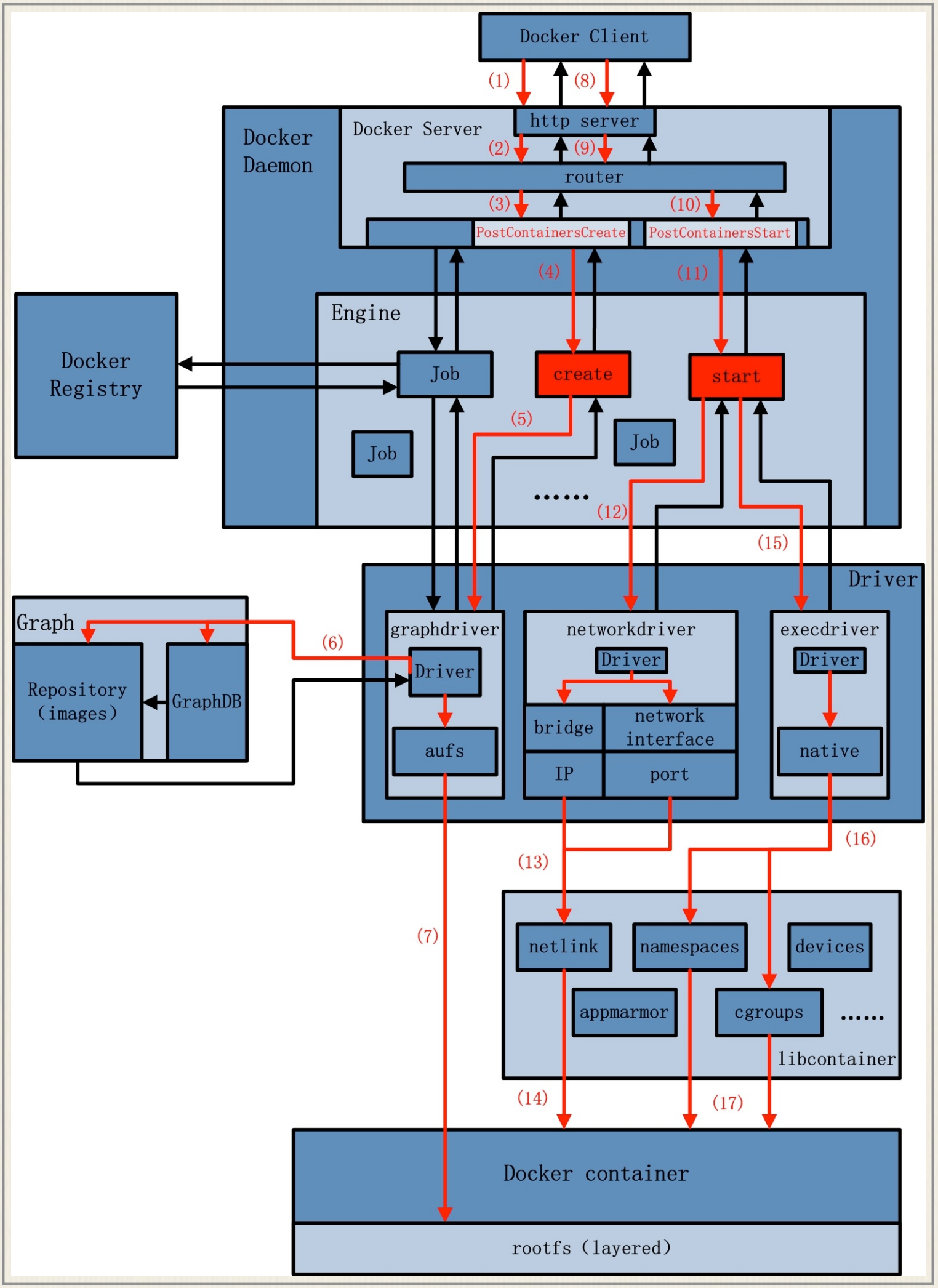


图5.2 docker run命令执行流程示意图

如图，图中标记的红色箭头表示docker run命令在发起后，Docker所做的一系列运行。以下逐一分析这些步骤。

(1) Docker Client接受docker run命令，解析完请求以及收集完请求参数之后，发送一个HTTP请求给Docker Server，HTTP请求方法为POST，请求URL为"/containers/create? "+"xxx"；

(2) Docker Server接受以上HTTP请求，并交给mux.Router，mux.Router通过URL以及请求方法来确定执行该请求的具体handler；

(3) mux.Router将请求路由分发至相应的handler，具体为PostContainersCreate；

(4) 在PostImageCreate这个handler之中，一个名为"create"的job被创建，并开始让该job运行；

(5) 名为"create"的job在运行过程中，执行Container.Create操作，该操作需要获取容器镜像来为Docker容器创建rootfs，即调用graphdriver；

(6) graphdriver从Graph中获取创建Docker容器rootfs所需要的所有的镜像；

(7) graphdriver将rootfs所有镜像，加载安装至Docker容器指定的文件目录下；

(8) 若以上操作全部正常执行，没有返回错误或异常，则Docker Client收到Docker Server返回状态之后，发起第二次HTTP请求。请求方法为"POST"，请求URL为"/containers/"+container_ID+" /start"；

(9) Docker Server接受以上HTTP请求，并交给mux.Router，mux.Router通过URL以及请求方法来确定执行该请求的具体handler；

(10)mux.Router将请求路由分发至相应的handler，具体为PostContainersStart；

(11)在PostContainersStart这个handler之中，名为"start"的job被创建，并开始执行；

(12)名为"start"的job执行完初步的配置工作后，开始配置与创建网络环境，调用networkdriver；

(13)networkdriver需要为指定的Docker容器创建网络接口设备，并为其分配IP，port，以及设置防火墙规则，相应的操作转交至libcontainer中的netlink包来完成；

(14)netlink完成Docker容器的网络环境配置与创建；

(15)返回至名为"start"的job，执行完一些辅助性操作后，job开始执行用户指令，调用execdriver；

(16)execdriver被调用，初始化Docker容器内部的运行环境，如命名空间，资源控制与隔离，以及用户命令的执行，相应的操作转交至libcontainer来完成；

(17)libcontainer被调用，完成Docker容器内部的运行环境初始化，并最终执行用户要求启动的命令。

6 总结

本文从Docker 1.2的源码入手，分析抽象出Docker的架构图，并对该架构图中的各个模块进行功能与实现的分析，最后通过两个docker命令展示了Docker内部的运行。

通过对Docker架构的学习，可以全面深化对Docker设计、功能与价值的理解。同时在借助Docker实现用户定制的分布式系统时，也能更好 地找到已有平台与Docker较为理想的契合点。另外，熟悉Docker现有架构以及设计思想，也能对云计算PaaS领域带来更多的启发，催生出更多实践 与创新。

原文链接：http://www.infoq.com/cn/articles/docker-source-code-analysis-part1?from=timeline&isappinstalled=0#10006-weixin-1-6358-0629b82e8bd20c82f766611c23eca2f9

前端学习之iOS开发（二）

作者：liuzq

本文是继上一篇《前端学习之iOS开发（一）》的续集，上一篇属于iOS开发的入门篇，主要内容为前端学习iOS开发的优势和对比学习。本文的内容为介绍在新的项目《有道口语大师》中的实际应用，主要包括第三方类库管理工具pods、sqlite、文件操作、网络请求、引导动画等在项目中的实践。

一、第三方类库管理工具CocoaPods

CocoaPods作为iOS开发的第三方类库管理工具，可以很好的解决第三方类库的检索、安装、更新等操作，功能与nodejs中的npm类似。

CocoaPods的安装和使用如下：

1、安装好Ruby环境

2、下载和安装命令(速度较慢)：sudo gem install cocoapods

3、在与.xcodeproj同级目录下建立Podfile文件，文件内容如下：

```
platform :ios, '6.1'
#3.忽略Pods所有警告
inhibit_all_warnings!
#网络请求
pod 'AFNetworking', '~> 2.2.4'

pod 'JSONKit',          '~> 1.4'
pod 'VKFoundation',     '~> 0.1.1'
pod 'Reachability',     '~> 3.1.1'
pod 'ASIHTTPRequest'
pod 'RegexKitLite'
pod 'NSData+Base64'
pod 'UIDeviceIdentifier'
pod 'ZipArchive'
pod 'MBProgressHUD', '~> 0.8'
```

文件中的内容用于指示下载哪些类库及指定类库版本号，例如，下载版本2.2.4的AFNetworking，不写版本的时候默认下载最新的版本，文件的意义同npm中的package.json。

使用 pod install 下载类库，使用该命令后会根据Pod-file中的列表下载需要的类库。

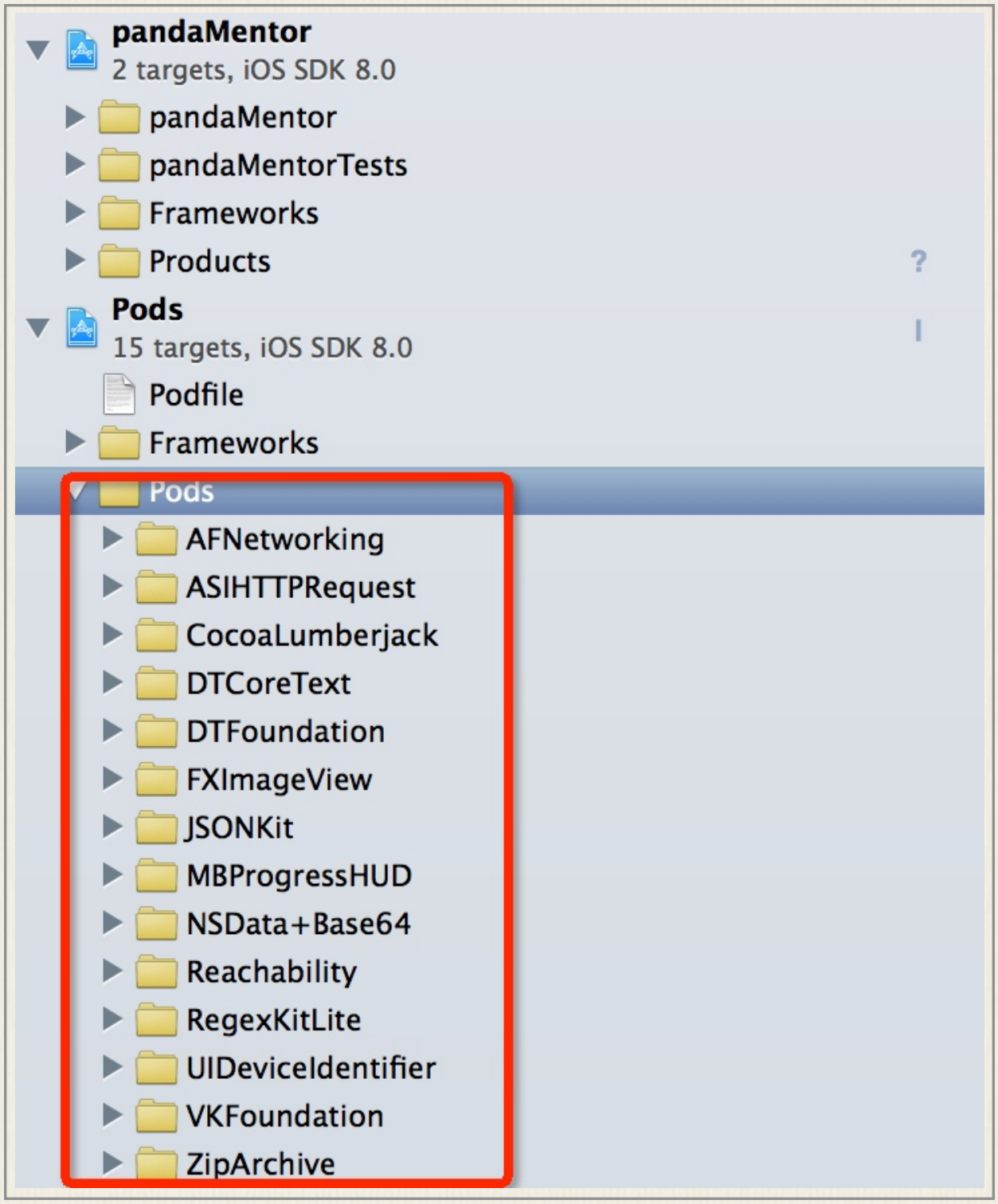
类库下载过程中截图：

```
-bash-3.2$ pod install
Analyzing dependencies
Downloading dependencies
Using AFNetworking (2.2.4)
Using ASIHTTPRequest (1.8.2)
Using CocoaLumberjack (1.7.0)
Using DTCoreText (1.6.13)
Installing DTFoundation 1.7.3 (was 1.7.2)
Installing FXImageView 1.3.5 (was 1.3.3)
Using JSONKit (1.5pre)
Using MBProgressHUD (0.9)
Using NSData+Base64 (1.0.0)
Using Reachability (3.1.1)
Installing RegexKitLite 4.0 (was 4.0)
Using UIDeviceIdentifier (0.4.2)
Using VKFoundation (0.1.1)
Using ZipArchive (1.3.0)
```

4、类库下载结束后会有如下提示

```
[!] From now on use `CocoaPodsDemo.xcworkspace`
```

.xcworkspace文件是使用pod install后生成的新文件，在我们项目的根目录下，接下来就是使用.xcworkspace文件来打开我们的项目了，而不是使用.xcodeproj。双击.xcworkspace打开项目文件后会看到如下的目录结构:



其中红色部分就是我们通过pods下载的类库了，以AFNetworking类库为例，可以通过#import “AFNetworking.h”引入下载后的类库。

另外CocoaPods支持搜索命令，例如在终端中输入命令‘pod search AFNetworking’，便可以获得AFNetworking类库的各版本信息，这一点和命令‘npm search ’功能相同。

5、 CocoaPods使用总结，例如Podfile中的列表，大家可以看到下载了很多类库，但如果下载后的类库都添加到svn版本库中，这无疑会使我们的项 目过于庞大，并且不利于其他同事checkout和管理，但我们可以不将下载后的类库添加到svn版本库中，只需添加PodFile到版本库中，这样需要

checkout项目的同事就可以根据PodFile了解项目中使用的类库并通过pod命令下载依赖的类库。

关于CocoaPods的详细介绍和其他命令可以参见：https://github.com/CocoaPods/CocoaPods/wiki和http://cocoapods.org/。

最后前端开发者是不是觉得CocoaPods与npm有异曲同工之妙呢。

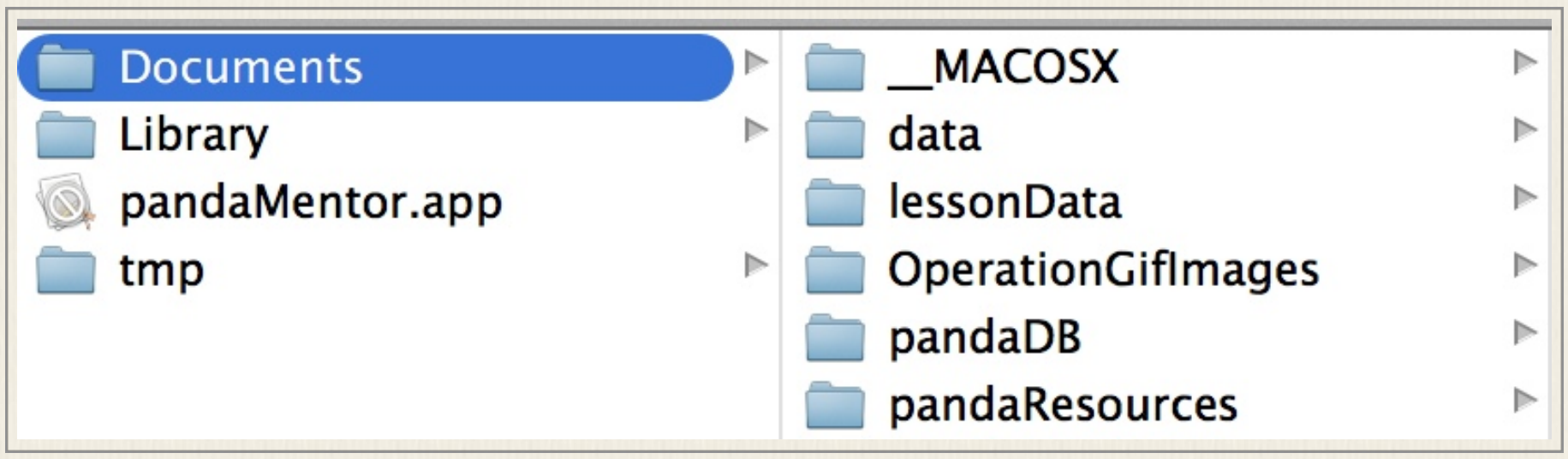
二、文件管理

1、NSBundle和NSDocumentDirectory

对于App有两个存储文件的目录结构，分别为：NSBundle和NSDocumentDirectory 。

NSBundle 目录中包含了程序会使用到的资源（图像、音频、视频、json文件等），这些资源都是随app第一次安装时进入到用户的手机中，并且存贮在app的安装包中。NSBundle目录是只读目录，也就是说在用户安装NSBundle后，里面的资源文件就不可修改了。所以如果资源文件是存储在NSBundle目录中，app上线后就算修改一个图片资源也要发布新的app版本。

NSDocumentDirectory是用户手机中应用程序沙盒中的目录，可用于存储在用户安装app后动态生成的文件，例如数据库文件，用户头像，从网络下载的其他资源等。下图为iOS模拟器中看到的《有道口语大师》应用程序沙盒中的目录结构：



其中data文件夹下存储的是语音评分的模型文件，lessonData文件夹下用于存储app中使用的数据包（json文件、图片、音 频），OperationGifImages用于存储app中运营活动的图片文件，pandaDB用于存储用户数据库和头 像，pandaResources用于存储即时更新的图片资源，从文件夹的描述上来看，大家应该明白NSDocumentDirectory目录与NSBundle目录的区别在于，前者用于存储动态更新的文件，因为前者是可以读写的，所以数据库文件和即时更新的文件都可以在这里存储。

在应用程序沙盒中除了Documents目录，还有Library目录，及tmp目录，但这两个目录都只适合临时存储。另外在向Documents目录下存储文件的时候一定要建立文件夹不要让文件散落到目录下，不然提交app的时候会被appleStore驳回。

下面简单介绍下如何获取到NSBundle和Documents目录，由于都是在api中有详细的讲解，这里只是想让初学者有个简单的了解：

（1）、获取NSBundle目录下的名为lessonData的压缩文件：

```
NSString *zipPath = [[NSBundle mainBundle] pathForResource:@"lessonData" ofType:@"zip" inDirectory:@"/"];
```

（2）、获得Documents根目录：

```
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
NSString *documentDirectory = [paths objectAtIndex:0];
```

2、文件操作

这里不会重点讲文件的增删改查，对于增删改查操作只是一套api而已，读者可自行深入学习。这里主要讲为什么要将 NSBundle中的部分资源文件拷贝到NSDocumentDirectory目录。这样可以让读者设计app的时候有更多的考虑。先看下我们的需求， 在用户随app安装的时候会在NSBundle目录下存储一个lessonData.zip文件，该文件中包含json、图片、音频等资源文件。而且这些 资源文件在app上线后会经常替换的，显然存在NSBundle目录下是不能达到即时更新的效果，所以我们要在app安装过程

中将 lessonData.zip文件解压到NSDocumentDirectory，这样做的好处是当json文件或者图片资源等有更新的时候可以通过网络 请求立即更新NSDocumentDirectory中对应的文件。代码如下：

```
//解压随app内置的lesson数据文件并拷贝到document下
 - (void)unzipFile {
     NSString *zipPath = [[NSBundle mainBundle] pathForResource:@"lessonData" ofType:@"zip" inDirectory:@"/"];
     NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
     NSString *documentDirectory = [paths objectAtIndex:0];
     ZipArchive *zip = [[ZipArchive alloc] init];
     if ([zip UnzipOpenFile:zipPath]) {
         [zip UnzipFileTo:[documentDirectory stringByAppendingPathComponent:@"/"] overWrite:YES];
         [zip CloseZipFile2];
     }
 }
```

其中ZipArchive为用于解压的类库，可参见https://github.com/mattconnolly/ZipArchive。这里有个需要注意的地方是调用UnzipOpenFile打开文件之后一定要调用CloseZipFile2关闭文件，否则会引起内存泄露。

对于文件的增删改查可以通过下面的代码稍作了解，其中NSFileManager是Foundation库中负责处理文件的类。

（1）判断文件是否存在，不存在时创建：

```
    if (![[NSFileManager defaultManager] fileExistsAtPath: pandaResources]) {
        [[NSFileManager defaultManager] createDirectoryAtPath:pandaResources withIntermediateDirectories:NO attributes:nil error:
            nil];
    }
```

（2）判断文件是否存在，存在时删除：

```
            if ([[NSFileManager defaultManager] fileExistsAtPath: toPath]) {
                [[NSFileManager defaultManager] removeItemAtPath:toPath error:nil];
            }
```

（3）循环拷贝NSBundle目录下文件到NSDocumentDirectory目录：

```objc
//  拷贝勋章图片
    NSString *honorPicDir = [[[NSBundle mainBundle] bundlePath] stringByAppendingPathComponent:@"Honors"];
    NSArray *honorPicContent = [[NSFileManager defaultManager] contentsOfDirectoryAtPath:honorPicDir error:nil];

    for (int i=0; i<honorPicContent.count; i++) {
        NSString *itemPath = [honorPicDir stringByAppendingPathComponent:[NSString stringWithFormat:@"/%@",honorPicContent[i]]];
        NSString *toPath = [pandaResources stringByAppendingPathComponent:[NSString stringWithFormat:@"/%@",honorPicContent[i]]];

        if ([[NSFileManager defaultManager] fileExistsAtPath: toPath]) {
            [[NSFileManager defaultManager] removeItemAtPath:toPath error:nil];
        }
        [[NSFileManager defaultManager] copyItemAtPath:itemPath toPath:toPath error:&err];
    }
```

图中最后一条语句便是拷贝方法，但需要在拷贝前先判断目标文件是否已经在目标目录，如果文件已存在时进行拷贝会引发错误。

（4）要获取json文件中的内容可以通过如下方法：

```objc
        NSString *jsonDataFile = [pandaResources stringByAppendingPathComponent:[NSString stringWithFormat:@"/%@",levelJsonFile]];
        NSData *data = [NSData dataWithContentsOfFile:jsonDataFile];
        NSError *error=nil;
        self.jsonData = [NSJSONSerialization JSONObjectWithData:data options:0 error:&error];
```

三、网络请求（AFNetworking）

对于网络请求前端开发者应该很了解了，前端中可以使用ajax来实现异步请求。这里使用的是AFNetworking库。AFNetworking 是ios和mac os x下的网络框架，它是构建在Foundation URL Loading System之上的，封装了网络的抽象层，可以方便的使用，AFNetworking是一个模块化架构，拥有丰富的api。支持当前网络连接情况判断、 GET、POST、POST Multi-Part格式的表单文件上传、下载及断点续传等操作。

分别用ajax和AFNetworking库实现POST请求的对比：

ajax:

```javascript
//调用ajax请求
$.ajax({
    type: "POST",    //设置请求类型为POST
    url: "some.php",     //请求地址
    data: "name=John&location=Boston",  //设置POST参数
    success: function(msg){      //成功回调
        alert( "success msg: " + msg );
    }
    fail:function(error){      //失败回调
        alert( "Error: " + error );
    }
});
```

AFNetworking：

```objectivec
//创建AFHTTPRequestOperationManager对象
AFHTTPRequestOperationManager *manager = [AFHTTPRequestOperationManager manager];
//定义POST参数
NSDictionary *parameters = @{@"foo": @"bar"};
//使用AFHTTPRequestOperationManager发送POST请求
[manager
    POST:@"http://example.com/resources.json"    //设置请求地址
    parameters:parameters    //设置请求参数
    //请求成功代码块
    success:^(AFHTTPRequestOperation *operation, id responseObject)
    {
        //operation中封装了请求的状态码（200、302等），请求返回的内容类型（json、plist、html等）
        //responseObject为请求响应的内容
        NSLog(@"SUCCESS JSON: %@", responseObject);
    }
    //请求失败代码块
    failure:^(AFHTTPRequestOperation *operation, NSError *error)
    {
        NSLog(@"Error: %@", error);
    }
 ];
```

可以看出使用方法上与ajax并无太多区别，只不过AFNetworking的方法名称等过于复杂，并在成功和失败的处理函数上使用的是object-c特有的代码块方式(block)。

这里着重讲下项目中利用AFNetworking完成下载数据压缩包的例子：

影片 目录.1 Lorem Ipsum dolor amet, consectetur

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do tempor incididunt ut labore et dolore magna aliqua.

上面gif图中展示的边下载边显示下载进度的效果就是使用AFNetworking实现的，并且在下载前可以利用AFNetworking中的AFNetworkReachabilityManager先判断网络情况，然后在利用AFDownloadRequestOperation实现下载及 获取下载进度的功能。

判断网络情况：

```objc
//生成AFNetworkReachabilityManager（网络可达性监测）对象
AFNetworkReachabilityManager *afReach = [AFNetworkReachabilityManager sharedManager];
//设置网络状态发生改变时触发的代码块
[afReach setReachabilityStatusChangeBlock:^(AFNetworkReachabilityStatus status) {
    switch (status) {
        case AFNetworkReachabilityStatusNotReachable:{
            //无网络链接
            break;
        }
        case AFNetworkReachabilityStatusReachableViaWiFi:{
            //通过WiFi连接
            break;
        }

        case AFNetworkReachabilityStatusReachableViaWWAN:{
            //通过WWAN连接
            break;
        }
        default:
            break;
    }
}];
```

下载并获取下载进度：

```objc
//      构建request对象，并设置超时时间为10s
        NSMutableURLRequest *request = [NSMutableURLRequest requestWithURL:downUrl cachePolicy:
            NSURLRequestReloadIgnoringLocalAndRemoteCacheData timeoutInterval:10];
//      生成AFDownloadRequestOperation对象，其中filePath参数为下载成功后文件的存储路径
        missionRequest = [[AFDownloadRequestOperation alloc] initWithRequest:request targetPath:filePath shouldResume:YES];
        missionRequest.responseSerializer = [AFJSONResponseSerializer serializer];
        missionRequest.responseSerializer.acceptableContentTypes = [NSSet setWithObject:@"application/zip"];
//      设置下载过程函数
        [missionRequest setProgressiveDownloadProgressBlock:^(AFDownloadRequestOperation *operation,NSInteger bytesRead, long long
            totalBytesRead, long long totalBytesExpected, long long totalBytesReadForFile, long long
            totalBytesExpectedToReadForFile) {
                NSLog(@"已下载%f",totalBytesReadForFile/(float)totalBytesExpectedToReadForFile);
        }];


//      下载成功和失败的处理函数
        [missionRequest setCompletionBlockWithSuccess:^(AFHTTPRequestOperation *downOperation, id downResponseObject) {
            NSLog(@"下载成功！");

        } failure:^(AFHTTPRequestOperation *downOperation, NSError *downError)
        {
            NSLog(@"Error: %@", downError);
        }];
//      开始发送下载请求
        [missionRequest start];
```

下载处理的使用和ajax方法也很类似，只是多了一个用于获取下载进度的setProgressiveDownloadProgressBlock方法设置。

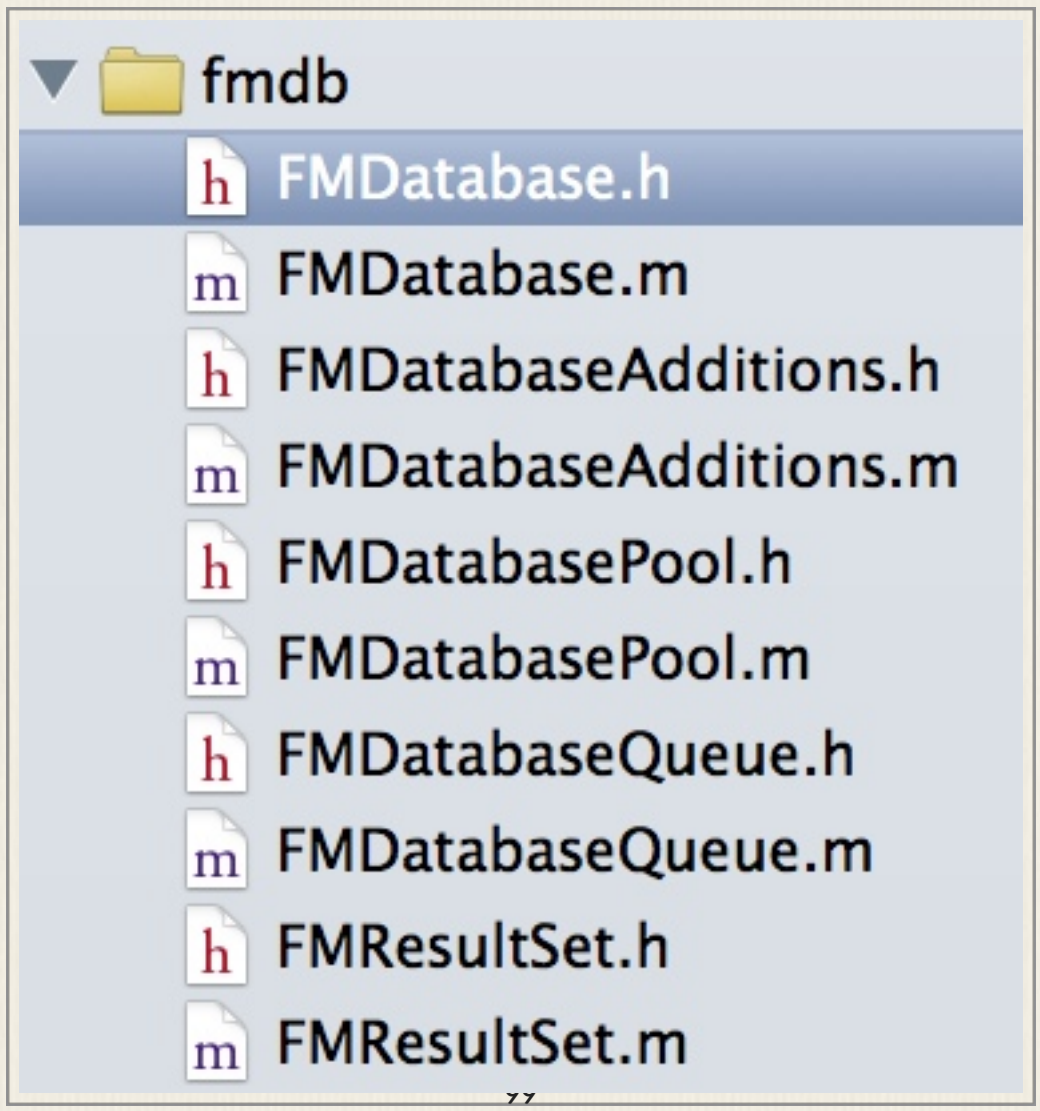
AFNetworking详细使用可参见：https://github.com/AFNetworking/AFNetworking

四、FMDatabase

在iOS开发中我们一般使用SQLite数据库，SQLite是一款轻型的数据库，是遵守ACID的关系型数据库管理系统，它的设计目标是嵌入式 的，而且目前已经在很多嵌入式产品中使用了它，它占用资源非常的低，在嵌入式设备中，可能只需要几百K的内存就够了。对于这种不需要存储在web服务器上的，被称为“SQLite”的文件型数据库目前已被广泛使用，前端开发者应该并不陌生，因为HTML5的本地存储中使用的也是SQLite数据库。即使没 有使用过也不必担心，对于iOS开发中使用的SQLite还是比较简单的，基本的sql语句就可以满足了 ，所以这里不需要有太多的数据库使用经验，对于一般项目来讲，大学的知识也就足够了。iOS中原生的SQLite API在使用上非常不便。于是，就出现了一系列将SQLite API进行封装的库，例如FMDB、PlausibleDatabase、sqlitepersistentobjects等，FMDB是一款简洁、易用的封装库，在《有道口语大师》项目中也是使用了FMDB。这里进行简单的介绍。

在下载FMDB文件后，工程中必须导入如下文件，并使用 libsqlite3.dylib 依赖包。

FMDB目录结构:



添加libsqlite3.dylib依赖包：

对于FMDB目录中的这三个文件做下功能描述：

FMDatabase： 创建SQLite数据库

FMResultSet： 获取执行sql查询后的结果集

FMDatabaseQueue： 在多个线程执行查询和更新时会使用这个类。

例如创建数据库：

db = [FMDatabase databaseWithPath:database_path];其中database_path可以是第二小节讲到的应用程序沙盒中documents下的一个路径。

打开和关闭数据库：[db open]、[db close]

创建表：

```
78  -(void)createTable{
79      if (self) {
80          if ([db open]) {
81              if (![db tableExists:@"lesson_info"]){
82                  [db executeUpdate:installLessonSQL];
83              }
84              if (![db tableExists:@"user_info"]){
85                  [db executeUpdate:installUserSQL];
86              }
87              if (![db tableExists:@"score_info"]){
88                  [db executeUpdate:installScoreSQL];
89              }
90              [db close];
91          }
92      }
93  }
```

可以使用FMResultSet对象的resultDictionary将结果集转为Dictionary：

```
296         rs = [db executeQuery:@"SELECT * FROM lesson_info where id = ?",id];
297         if ([rs next]){
298             data = [rs resultDictionary];
299         }
```

通过intForColumn、stringForColumn、longForColumn等方法获取某一字段的值：

```
324         rs = [db executeQuery:@"SELECT firstStep FROM user_info where uid = ?",[defaults objectForKey:@"userid"]];
325         if ([rs next]){
326             firstStep = [rs intForColumn:@"firstStep"];
327         }
```

在使用过程中一定要注意[db open]和[db close]的配对使用，否则会造成内存泄露引起app崩溃。尤其注意某些数据库操作函数的嵌套使用。例如：

```
-(void)saveMissionScore:(NSString*)mid score:(NSNumber*)score{
    FMResultSet *rs;
    if ([db open]) {
        //解锁下一个mission
        [self unlockMissionId:mid];
        rs = [db executeQuery:@"SELECT * from lesson_info WHERE id = ?",mid];
        [db close];
    }
}
-(void)unlockMissionId:(NSString*)id{
    if ([db open]) {
        [db executeUpdate:@"update  lesson_info set lockStatus=1 where id=?",id];
        [db close];
    }

}
```

这段代码存在两个问题，在saveMissionScore:(NSString*)mid score:(NSNumber*)score方法中已经执行了[db open]，但在unlockMissionId方法中又执行了[db open]，另外在unlockMissionId中还执行了[db close]，在unlockMissionId方法执行完毕时数据库已经处于关闭状态了，这时又去执行了sql操作，前面这两点都会引起数据库操作异常 从而引起app崩溃。所以一定要确保[db open]和[db close]的配对使用。

开发者可以使用SQLite Professional、火狐浏览器的SQLite Manager等工具进行sql语句的验证及查看数据库文件。

对于数据库的增删改查语句这里就不展开介绍了，可以通过https://github.com/ccgus/fmdb和http://www.sqlite.org/inmemorydb.html去查看。

五、引导动画

目前很多app在启动后都会有一个引导动画，很多比较复杂和绚丽的动画会在一开始就吸引住用户。这一节主要是想通过app启动时的一个引导动画来学习iOS开发中的UIScrollView和动画的实现过程。先看下《有道口语大师》启动的引导动画：

影片 目录.2 **Lorem Ipsum dolor amet, consectetur**

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do tempor incididunt ut labore et dolore magna aliqua.

实现分析：

1、对于UIScrollView的理解，前端人员可以理解为一个未设置overflow：hidden的div，当内容溢出容器后可以滚动显示出溢出的内容。

2、当我们想要知道div的滚动距离的时候肯定要监听div的scroll事件，在iOS的这个例子中当实现了UIScrollView的 UIScrollViewDelegate协议后，我们就可以监听到UIScrollView的一切动作，包括横向、纵向滚动，及滚动的开始和结束。

3、通过如下设置便可实现UIScrollView的分页，假设设置分为3页：

```
33          winsize = self.frame.size;
34          guideScrollView.pagingEnabled = YES;
35          guideScrollView.scrollEnabled = YES;
36          guideScrollView.contentSize = CGSizeMake(winsize.width*3, guideScrollView.frame.size.height);
37          UIPageControl *guideControl;
38          guideControl.numberOfPages = 3;
39          [guideControl setCurrentPage:0];
```

其中guideControl是动画下面显示的分页器。

4、在前面3点介绍的iOS相关内容之后，剩下的就是我们前端擅长的动画操作了，只要实现UIScrollViewDelegate的scrollViewDidScroll:(UIScrollView *)scrollView方法，就可以监听到滚动的长度，该长度通过方法中的scrollView.contentOffset.x获得，然后通过该值来 做相应的动画变化。其中涉及到的动画效果有CGAffineTransformMakeScale（缩放）、CGAffineTransformTranslate（移动）、CGAffineTransformRotate（旋转），这几个动画效果在CSS3中 也有对应的实现方式，分别为：transform:scale(0.8,1)、translate(50px,50px)、 transform:rotate(45deg),所以对于前端开发者并不难理解这些动画实现，接下来我们是不是也可以在webApp中实现一些 native可以做的动画效果呢!
由于动画的代码还挺多，这里不贴出来了，可以看下如何通过scrollViewDidScroll:(UIScrollView *)scroll-View方法获取到横向滚动距离，代码如下：

```objc
- (void)scrollViewDidScroll:(UIScrollView *)scrollView
{
    //打印scrollView的横向滚动距离
    YDLog("%f",scrollView.contentOffset.x);
    //利用横向滚动距离与屏幕宽度计算出当前页码
    int page = guideScrollView.contentOffset.x / winsize.width;
    //设置分页器的页码
    guideControl.currentPage = page;

    //接下来可以利用横向滚动距离来做动画...
}
```

5、值得一说的是第二针的实现，读者仔细看第二针有很多小元素围绕着手机图片，如果这些小元素都拆分开再通过改变位置的方法实现显然不是最好的解决方案， 代码会很复杂。那最后选择的方案是用一张大图，大图中只包含围绕着手机的小元素，并且将图片通过CGAffineTransformScale放大到8倍 （这里的8倍是试出来的，主要是放大到足够大后保证小元素不会在屏幕上显示出来），滚动过程中再通过滚动的距离和图片放大后的宽度比例缩小图片，直到缩小 到0。

六、使用GCD实现多线程（Grand Central Dispatch）

这个小节主要讲项目中一个例子，运用GCD实现异步操作。关于GCD的概念如下：

Grand Central Dispatch或者GCD，是一套低层API，提供了一种新的方法来进行并发程序编写。从基本功能上讲，GCD有点像NSOperationQueue，他们都允许程序将任务切分为多个单一任务然后提交至工作队列来并发地或者串行地执行。

与HTML5中的Web Workers目的相同，我们使用多线程大多数目的都是为了将某些耗时较长的处理交给后台线程去运行。例如下面这段js代码：

```
var result=0;
for(var i=0;i<num;i++)
{
    result+=i;
}
console.log("和值为: "+result);
```

当num的值为100亿(不同的浏览器中有所差别)以上的值时，浏览器会弹出一个脚本运行时间过长的对话框，从而不得不终止当前计算。但如果将上述代码放 在一个单独的js文件中，并通过Web Workers去执行，并在执行完毕后通过postMessage方法将执行结果发送给主线程，这样无论num为多大的值都可以正常计算并且不会影响用户 的其他操作了。代码如下:

新建一个用于计算的sum.js:

```
onmessage = function(event)
{
    var num = event.data;
    var result = 0;
    for (var i=0; i<num; i++) {
        result += i;
    }
    //向主线程发送消息
    postMessage(result);
}
```

在主线程中创建子线程并调用sum.js计算:

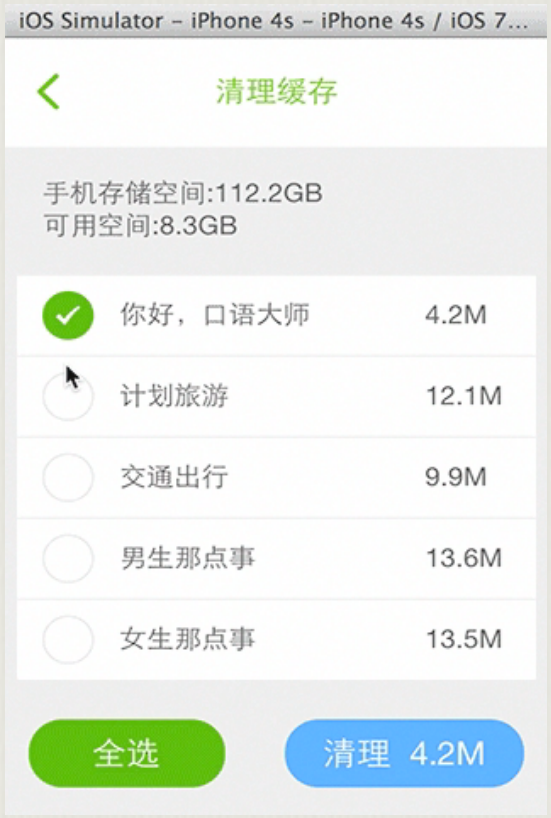
```
//创建执行运算线程
var worker = new Worker("sum.js");
//接收worker线程传递的计算结果
worker.onmessage = function(event)
{
    console.log("和值为: "+event.data);
}
//向worker线程传递参数
worker.postMessage(100000000000);
```

以上代码可用object-c改写为:

```
//创建后台执行线程，DISPATCH_QUEUE_PRIORITY_DEFAULT参数为线程执行的优先级
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    //耗时操作
    long long result=0;
    for (long long m=0; m<100000000000; m++) {
        result+=m;
    }
    //返回主线程
    dispatch_sync(dispatch_get_main_queue(), ^{
        NSLog(@"和值为: %lld",result);
    });
});
```

通过上面的对比例子之后，我们看下《有道口语大师》中实现清理app中的音频和图片资源的功能，并显示清理进度，实现效果如下:

影片 目录.3 Lorem Ipsum dolor amet, consectetur



Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do tempor incididunt ut labore et dolore magna aliqua.

读者可以根据下列代码及注释去理解，采用GCD的原因主要是删除文件过程是个耗时的操作，会阻塞UI线程，所以要创建一个异步线程在后台执行删除操作，再 删除操作执行完毕后回到UI主线程更新删除进度，最后当删除操作全部完成后回到主线程中更新tableview中的被删除行。

```objc
//  创建异步删除队列
    dispatch_async(dispatch_get_global_queue(0, 0), ^{
        //以下为处理耗时的删除操作
        NSMutableArray *indexPathArr = [[NSMutableArray alloc] initWithCapacity:1];
        int deletedCount = 0;
        for (int i=0; i<levelCachArr.count; i++) {
            YDCachLevelCellItem *cellItem = [levelCachArr objectAtIndex:i];
            NSString *id;
            if (cellItem.selected) {
                id = cellItem.levelId;
                //执行删除
                [YDCathMoudle removeLevelFolder:id];
                deletedCount +=1;
                NSIndexPath *indexPath = [NSIndexPath indexPathForRow:i inSection:1];
                [indexPathArr addObject:indexPath];
                //建立同步队列，并进入主线程实现更新删除的进度条
                dispatch_sync(dispatch_get_main_queue(), ^{
                    [blockSelf->deleteProgress setProgress:deletedCount/(float)selectedArr.count animated:YES];
                });
            }
        }
        //创建同步队列，实现tableview中被删除行的ui更新
        dispatch_sync(dispatch_get_main_queue(), ^{
            [blockSelf->win setHidden:YES];
            blockSelf->win = nil;
            NSArray *items = [self generateTableData];
            blockSelf->_items = items;
            [blockSelf.tableView beginUpdates];
            [blockSelf.tableView deleteRowsAtIndexPaths:indexPathArr withRowAnimation:UITableViewRowAnimationAutomatic];
            [blockSelf.tableView endUpdates];
            [blockSelf updateCleanBtn];
        });
    });
```

其中dispatch_async为开启一个异步操作，第一个参数是指定一个gcd队列，第二个参数是分配一个处理事物的代码块到该队 列，dispatch_get_global_queue(0, 0)，指用了全局队列，一般来说系统本身会有3个队列：global_queue，current_queue,以及main_queue。 其中第一个参数0的值也是DISPATCH_QUEUE_PRIORITY_DEFAULT值，即可以写成：
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)，除此之外还有其他数值：DISPATCH_QUEUE_PRIORITY_HIGH，DISPATCH_QUEUE_PRIORITY_LOW，通 过变量名成就可以理解出是设置异步队列的优先级，优先级的定义如下：

```objc
#define DISPATCH_QUEUE_PRIORITY_HIGH 2
#define DISPATCH_QUEUE_PRIORITY_DEFAULT 0
#define DISPATCH_QUEUE_PRIORITY_LOW (-2)
#define DISPATCH_QUEUE_PRIORITY_BACKGROUND INT16_MIN
```

通过以上例子，最终我可以将代码简化为如下图所示的结构：

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
    // 处理耗时操作的代码块...
    |
    //通知主线程刷新
    dispatch_async(dispatch_get_main_queue(), ^{
        //回调或者说是通知主线程刷新，
    });

});
```

关于iOS中的多线程，除了GCD，读者还可以了解下使用NSThread、NSOperation和NSOperationQueue来实现多线程。

七、链式调用

前端开发者应该都很熟悉和喜欢使用jQuery 的链式写法，例如如下代码：

```
//将id为title的元素内容设置为"test"，并将该元素显示出来
$("#title").text("test").show();
```

在iOS开发中我们也可以通过返回self的方式实现链式调用，例如新建一个YDDictQuerier类：

```objc
@interface YDDictQuerier : NSObject

//定义title和isHidden属性
@property (weak, nonatomic) NSString *title;
@property (nonatomic, assign) BOOL isHidden;

//声明设置title和是否隐藏方法
- (YDDictQuerier *)applyTitle:(NSString *)inTitle;
- (YDDictQuerier *)applyHidden:(BOOL)hidden;

@end
```

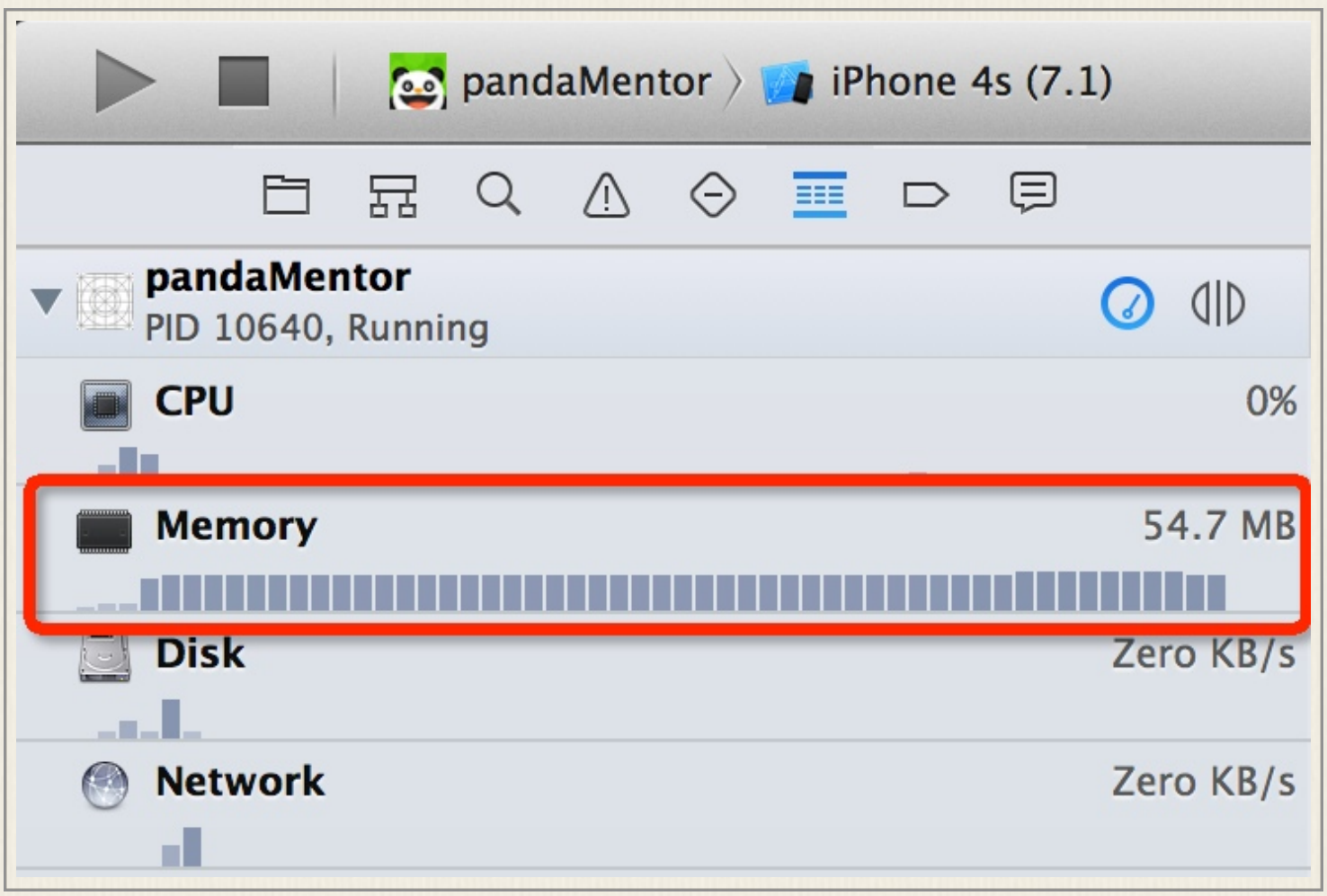
```objc
//设置title
- (YDDictQuerier *)applyTitle:(NSString *)inTitle
{
    self.title = inTitle;
    //返回self，实现链式调用
    return self;
}
//设置是否隐藏
- (YDDictQuerier *)applyHidden:(BOOL)hidden{
    self.isHidden = hidden;
    return self;
}
```

现在就可以像使用jQuery一样采用链式写法了，代码如下：

```objc
//生成YDDictQuerier对象
YDDictQuerier *dict = [[YDDictQuerier alloc] init];
//使用链式调用的方式设置title和是否隐藏属性
[[dict applyTitle:@"test"] applyHidden:NO];
```

八、内存监测与优化

本小节是讲在app启动后，如何通过Xcode随着不同的操作及app使用时间的增长来观察app所占用的内存。在前端开发中也有内存监测工具，例 如chrome浏览器为开发者内置的内存跟踪工具，可以通过开发者工具的Profiles标签调出，该工具可以拍下当前JS的heap快照，并可以看到closure、array、object等的内存使用情况。在前端开发中例如：闭包上下文绑定后没有释放，定时器的处理函数没有及时释放（没有调用clearInterval方法）等都会引起内存泄露。在iOS开发中多数情况下也是因为没有及时释放不使用的内存引起app崩溃或由于内存占用过高导致 app卡顿。当监测到app执行某一操作或到某一时间点的时候所占用的内存居高不下，并持续增长的话，基本确定app是有内存占用问题的。如下图在 xcode6中提供的app内存监测界面：



图中显示的是当我们进入第四节中讲到的app引导动画界面后的内存变化情况，还记得在引导页面我们引入了一张很大的图片吧，由于这张图片的原

因所以 内存会变的很高，起初遇到这个问题，首先想到的就是在离开引导页面的时候删除图片引用就可以恢复内存使用量到较低的状态，但并没用达到预期效果，原因是虽 然此时内存会居高不下，但这种由于引入大图片引起的内存过高是由于在引入图片后图片会缓存到内存中，这时即使使用代码强制删除图片引用，缓存中的图片也不 会立即释放的，但稍后iOS系统会在图片不使用的阶段自动释放掉。但如果内存居高不下，持续增长的时候，开发者可以在对应的页面或执行的某个操作处视具体 问题来查看是否有强引用、或某些耗时操作等需要优化，常出现的内存过高问题一般是在某一个controller或者view组件被移除或不需要引用的时候 对应的dealloc方法由于强引用或其他原因没有执行引起的，可以通过在dealloc方法中设置断点或者打印log的方式查看相应的 controller或view是否被释放，如果是强引用引起的视情况改为弱引用，或将耗时、耗内存的操作用上一节讲到的异步线程处理。

到这里本篇文章就结束了，主要介绍的是实现思路和类库的使用，并通过与前端开发进行对比学习加深理解和记忆，但并没有详细到代码级别，感兴趣的同学可以自己边看api边写demo来进一步学习。

原文链接：http://techblog.youdao.com/?p=1023

Web开发者和设计师必须要知道的iOS 8 十个变化

译者：罗磊

喜大普奔，喜极而泣，喜当爹，随着iPhone 6和iPhone 6 plus的上市，ios 8终于在上周推送更新了。新的设备，新的分辨率。接下来这篇文章介绍下 iOS 8有哪些变化。

概述

- 简介 iOS 8 上的 Safari 的更新
- iPhone 6 和 iPhone 6 Plus
- 新 Api 支持
- Safari 新功能和支持
- iOS 8 原生优化
- Safari 插件
- 新的设计
- 视频增强
- iOS 8上的JS
- Bug 和问题

已经习惯了苹果官方的高冷，这次，苹果依旧没有更新任何与 Safari 或者 iOS 相关的文档，所以下面的所有数据和资料都是基于我自己的测试和 WWDC 上公布的信息。

iOS 8 上的 Safari

- 支持HTML5新APIs: WebGL (3D canvas), IndexedDB, Navigation Timing API， Crypto API
- 混合应用: 更快的、优化的WebView
- 支持滚动 Scroll 事件:终于支持了!
- 视频播放: 全屏API，元数据API
- HTML模板元素
- Safari 插件:原生App可以以插件的形式读取网页DOM
- 图片:支持Image Source Sets和动态PNGAPNG
- CSS：支持Shapes，支持小数单位
- 浏览器自动填写表单（支持信用卡调用摄像头扫描）
- 网页和本地应用交互：登录数据共享
- EcmaScript 6： 部分支持
- SPDY:支持谷歌家的新网络协议了
- 文件上传失效了(这是Bug)
- 移除了minimal-ui属性
- 支持Yosemite上的远程调试

相比其他移动端上的浏览器，iOS 8并没有支持有些功能:

- dp单位的Media queries
- getUserMedia:访问本地硬件设备，捕获音频和视频的Api
- WebRTC:网页即时通信
- @viewport 声明
- Datalist
- WebP图片

iPhone 6 和 iPhone6 Plus

iPhone 6 和 iPhone6 Plus 是苹果继 iPhone 5 后的又一款不同尺寸和不同分辨率的设备。iPhone 6 的参数为4.7寸大小和750×1334物理分辨率的屏幕（dpi 值与iphone 5s 相同），iPhone 6 Plus则是5.5寸和1080×1920分辨率(401 dpi)的屏幕。不走寻常路的苹果给这两分辨率取名叫Retina HD屏，嗯哼，比Retina多了一个HD。

对于web开发者来说，不同的不仅仅是尺子上的大小。还包括默认viewport（关乎 width=device--width的设置），像素比（关乎高清图片的应用），icon图标大小和登录页的图片大小。

	iPhone 6	iPhone 6 Plus
尺寸	4.7”	5.5”
Viewport’s device-width (in CSS pixels)	375	414
Viewport’s device-width (Android设备同分辨率参考)	360	400
Device Pixel Ratio 像素比	2	3(近似值)
Rendered Pixels 渲染像素 (默认 viewport size * dpr)	750×1334	1242×2208
Physical pixels 物理像素	750×1334	1080×1920

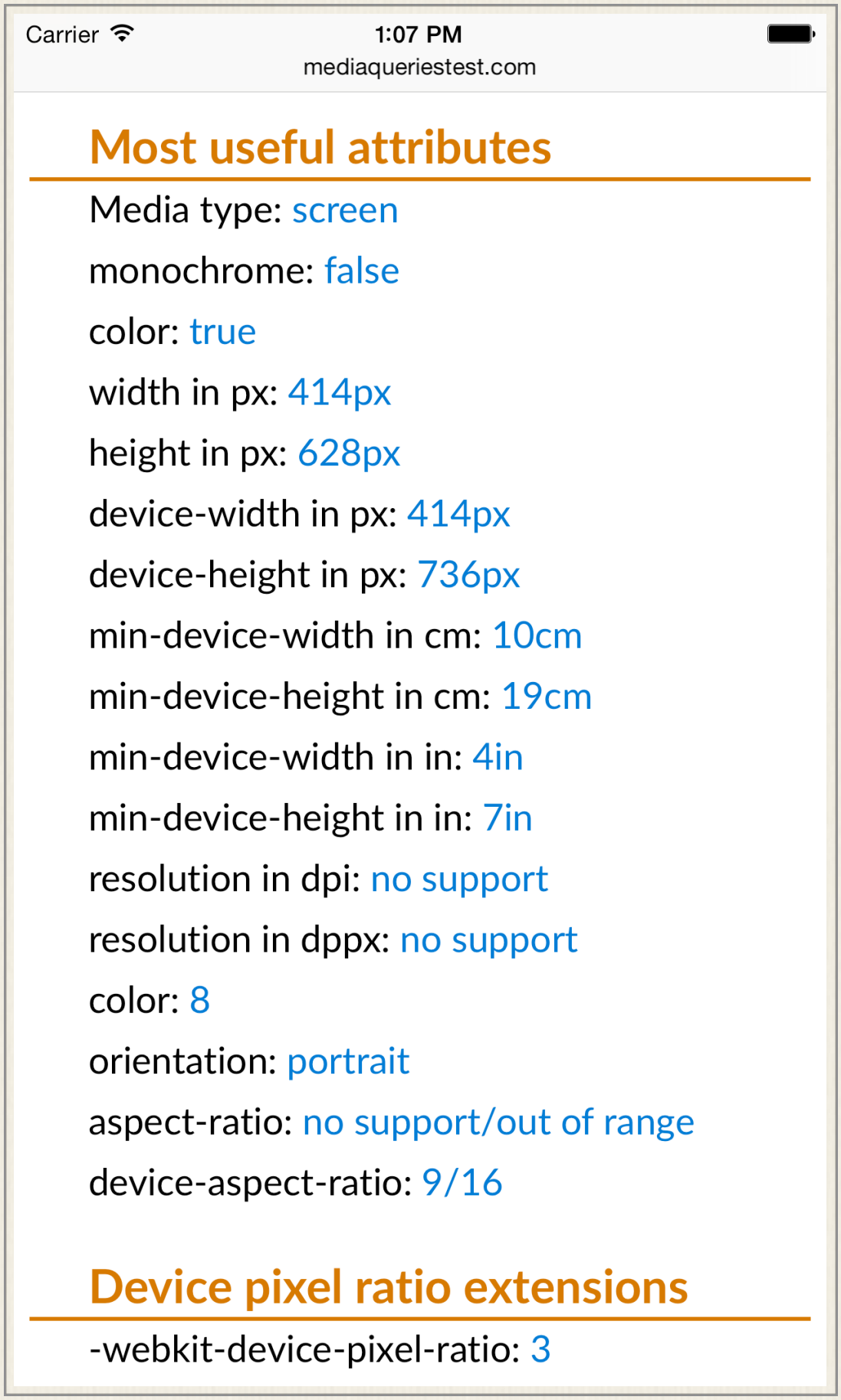
对于新 iPhone 的屏幕尺寸，推荐一篇文章"iPhone 6 Screens Demystified" 。

VIEWPORT SIZE

正在读这篇文章的你应该已经知道

直到上周，所有的 iPhone 和 iPod 使用的都是320px的屏幕宽度。iPhone 6 和 Plus 相比前代更加宽，给我们带来了更多的空间，苹果终于决定加宽浏览器宽度了。但是苹果奇葩的是使用了一套特殊的屏幕像素值。大部分4.7~5寸的安卓设备的viewport宽设为360px，iPhone 6上却是375px，大部分5.5寸安卓机器（比如说三星Note）的viewport宽为400，iPhone 6 plus 上却是十分怪异的414px （ㄟ(▔, ▔)ㄏ苹果你这样折腾是闹哪样啊）。这意味着相比同样尺寸的安卓机器，iPhone 6用户大概要少看4%的

内容。也许这并不是什么大问题，但是你也许还是得检查下你的网站是否适配。



适配新iPhone，你可以使用下面两段<meta>

<meta name="viewport" content="width=375">

<meta name="viewport" content="width=414">

设备像素比

iPhone 6与 iphone 5一样，像素比都是2，但是另一方面 iPhone 6 401的dpi真实的像素比值应该大约是2.60。为了解决这个问题，苹果又整了个新概念rendered pixels 渲染像素，如果像素比是3x,那么理论上一个css宽设为414px的屏幕应该拥有1242px的物理像素（现实中是1080px，小了13%）。

因此，如果你使用一个3x的图给高清的安卓设备，同样这样图也会适配iPhone 6 Plus 但是iPhone 的浏览器在渲染在屏幕之前首先会调整图片大小。

图标大小

iOS特有的图标大小，在 iPhone 6 plus上是180×180，iPhone 6 上则还是老的120×120。

适配iPhone 6 plus，则需要在中加上这段

<link rel="apple-touch-icon-precomposed" sizes="180x180" href="retinahd_icon.png">

启动图

如果你的webapp有一个启动图，那么你又得增加两行代码适配新iPhone 了。

iPhone 6对应的图片大小是750×1294，iPhone 6 Plus 对应的是1242×2148 。

<link rel="apple-touch-startup-image" href="launch6.png" media="(device-width: 375px)">

<link rel="apple-touch-startup-image" href="launch6plus.png" media="(device-width: 414px)">

UA探测

目前位置，所有升级到iOS 8的 iPhone都使用同样的UserAgent，所以我们暂时还没有办法在服务端判断这是什么设备，当然，通过JS和Media Queries我们还是可以通过技巧来判断的。

新的API

两个最重要的Api支持终于登录了iOS,分别是WebGL和IndexedDB，与此同时，Safari也开始支持Web Cryptography和Navigation Timing。

WebGL支持3D模拟，并且是浏览器默认开启。对于游戏开发者来说，这是一个好消息，更加丰富的交互和形式将在网页上出现。

你可以在微软的这个FishGl测试下 iOS 8 Safarai上进行3D渲染的效果。

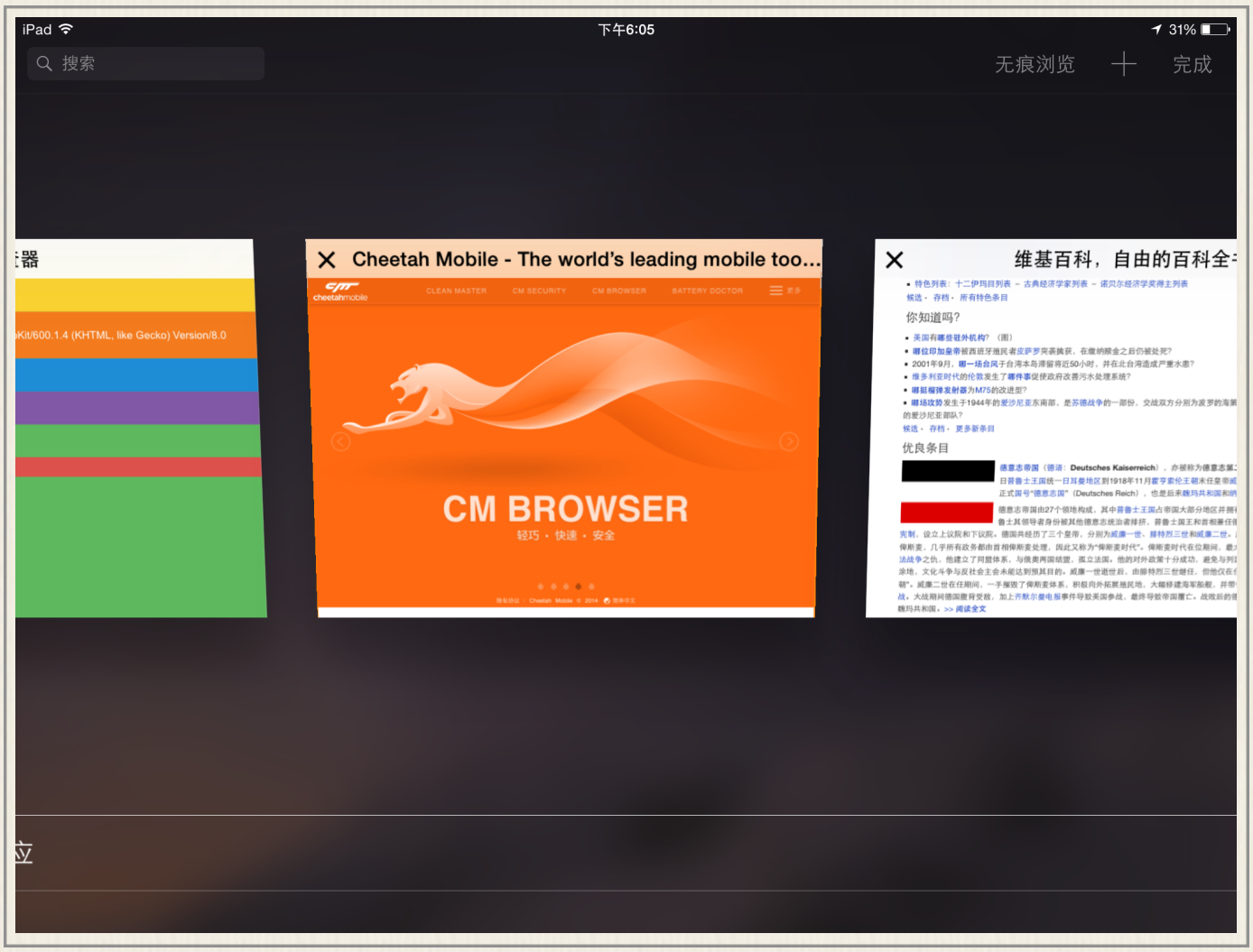
IndexdDB是W3C继起用WebSQL后推出的新的标准，随着 iOS支持 IndexedDB，我们能在不同的手机浏览器上使用同一套数据库API。

Navigation Timing API对于web性能优化来说是个好消息。通过这个API我们能过测量更加精准的加载渲染时间，优化网页的用户体验。

Safari新功能

缩放

iPhone处于横屏的时候,iPad（横竖均可），如果你用手指捏放屏幕（比如说你想放大网页），你会进入一个标签预览模式，用起来倒是挺方面，可是可能会与你在网页上使用的gesturechange事件所冲突，如果你要使用自定义缩放事件，首先爱你你得event.preventDefault()来阻止浏览器的默认事件。

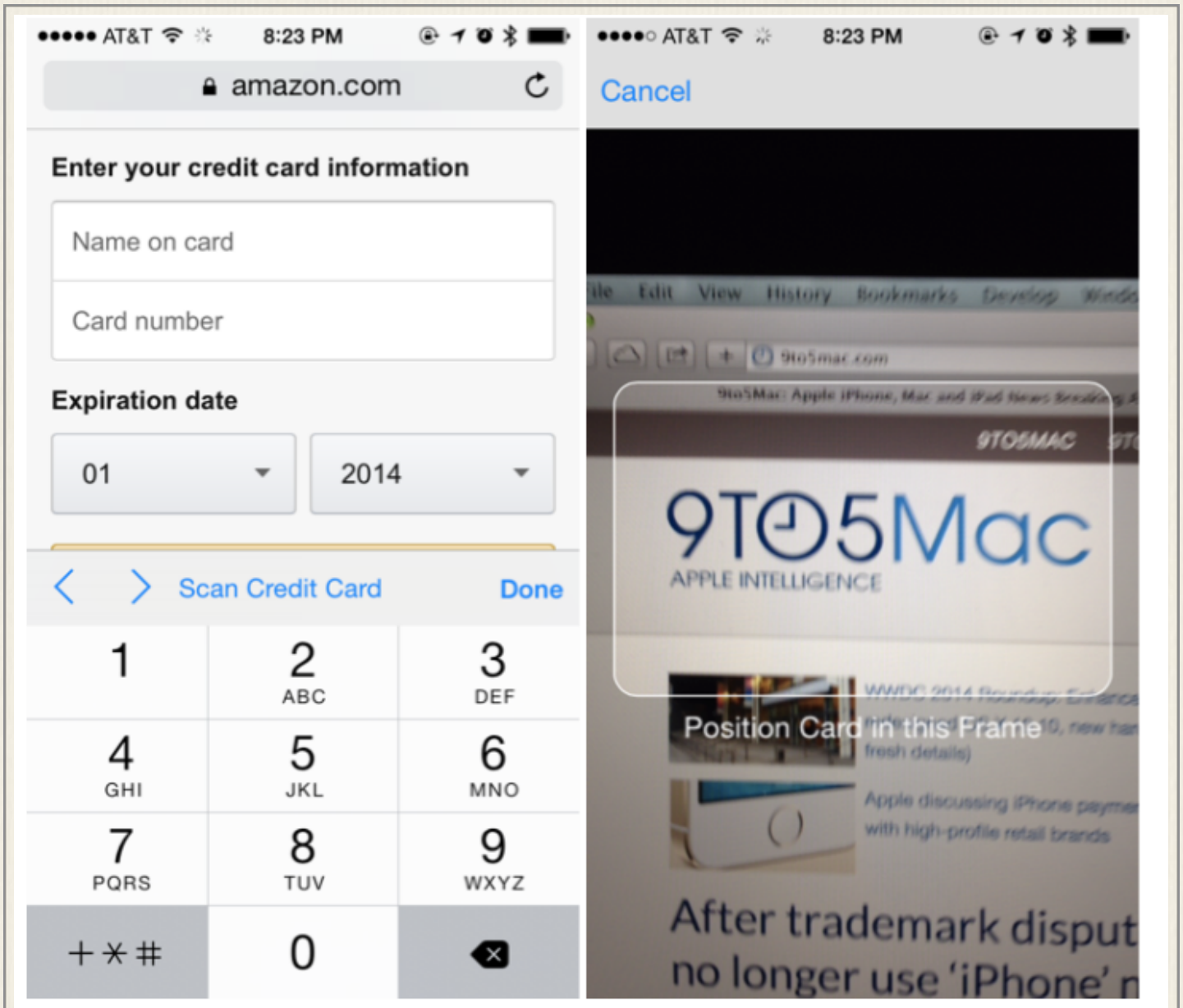


取消 MINIMAL-UI

六个月前，苹果推出了了Minimal UI mode《iOS 7.1的Safari为meta标签新增minimal-ui属性，在网页加载时隐藏地址栏与导航栏》，iOS 8更新后则又取消了这个，依旧高冷女神范，没说为什么。

表单自动填写和信用卡扫描功能

Safari如今支持自动补全表单，并且当Safari检测到你要填写的是信用卡的时候他会允许你开启摄像头直接扫描实体信用卡。



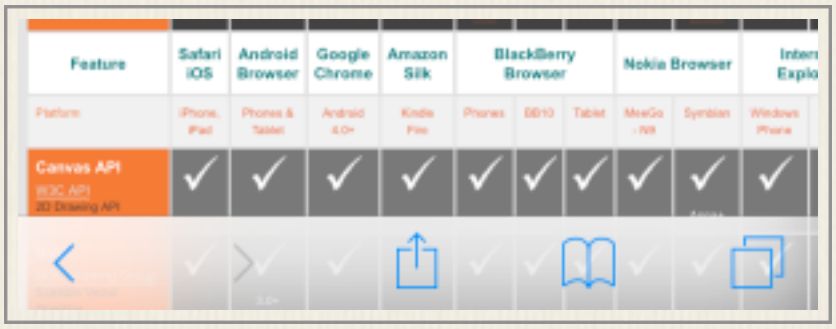
于此同时，Safari也支持autocomplete属性，参考latest spec。这意味着如果你在一个登录页面，Safari可以调用Keychain里的数据自动填写用户名帐号、密码。Luis Abreu写过一篇关于iOS 8安全和隐私相关的文章，推荐可以看看。

RSS!

如果你的网站提供了RSS订阅，iOS 8用户可以直接在浏览你网页的时候在书签栏打开他。就是那个@图标里面有个订阅的按钮，虽说有点小低调，但好歹还是有嘛 ﾉ(´ ▽ `)ﾉ。

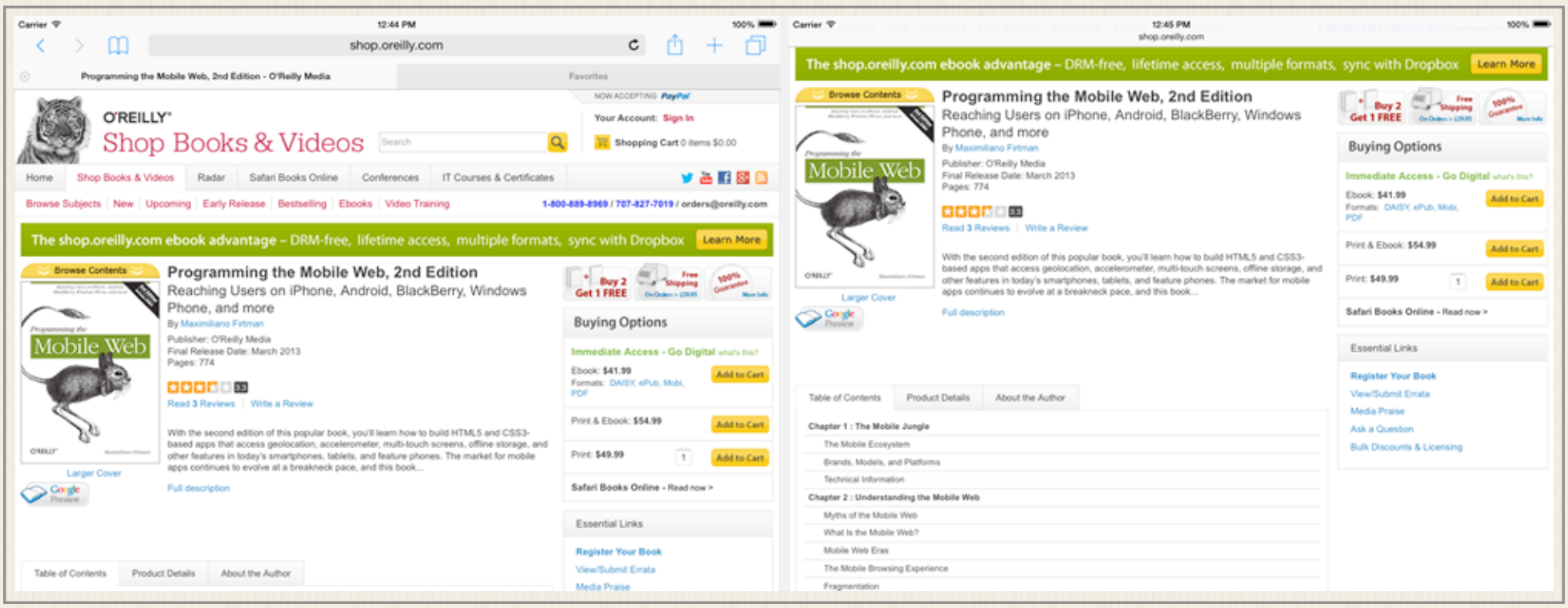
工具栏

现在地址栏和工具栏就变成半透明了。在 iOS 7上只有地址栏是半透明的。这意味着初次加载的时候网页可视区域变得更大（包含了底部的工具栏）。

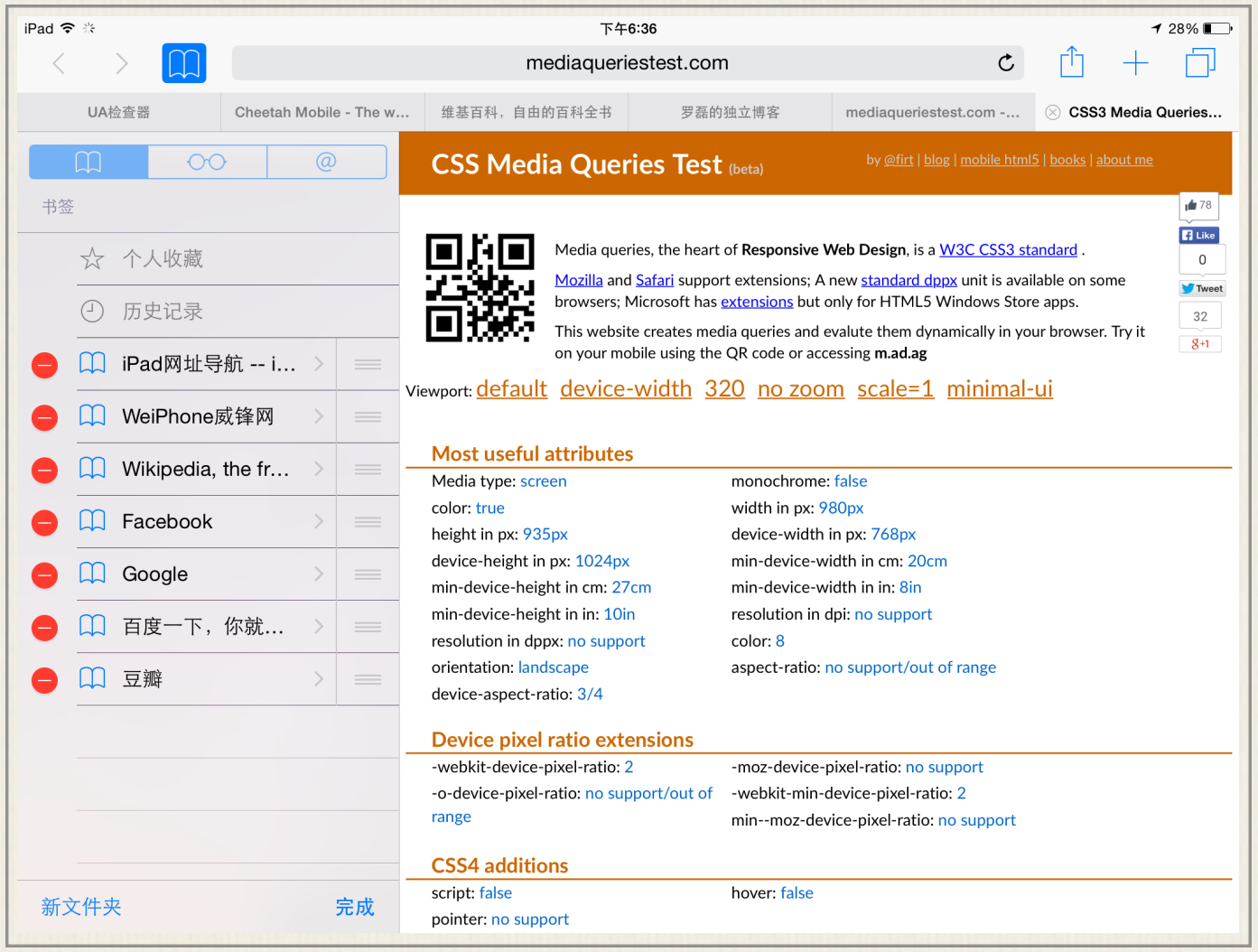


iPad上的更新

iOS 7以后，使用iPhone上的Safari浏览网页，滚动的时候会自动隐藏工具栏（ipad不会），iOS 8 ipad也有了这个细节上的变化（横竖都有）。不同的是，在横向模式的时候，iPad上面还有一小条，iPhone则是全屏。

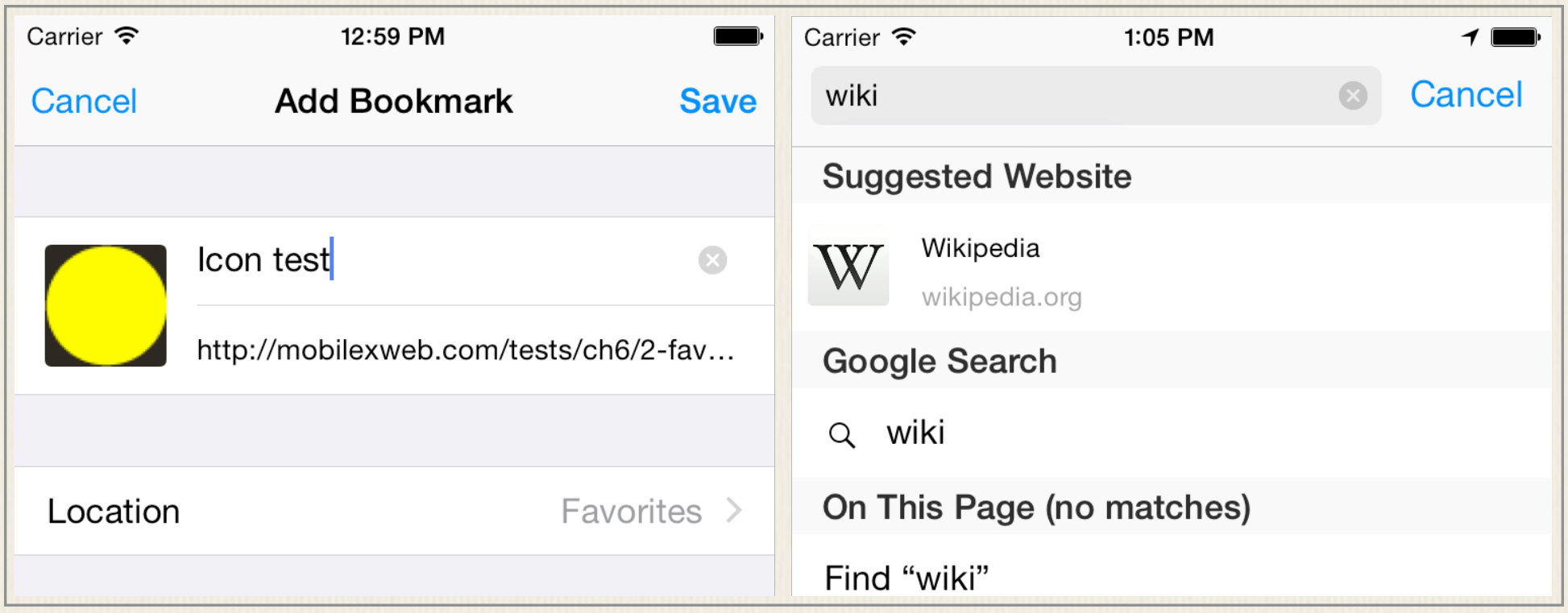


除此之外，Safari在横屏模式开启侧边栏会保持原有的viewport大小，只会改变aspect-ratio



书签icon和常用网站

Safari终于支持收藏夹里和书签里的网页附带网站的icon图标。当你在地址栏输入关键字搜索的时候，同样会展示网站的icon（ipad刚刚我测试了下没有）。



跨平台切换

如果你同时使用Mac和iPhone，当你在iPhone上浏览一个网页，你可以在你的Mac上（需要Yosemite）继续阅读（今年WWDC上专门演示了这个功能，还得等到Yosemite的正式更新）。

更像native的webapp

如果你想让用户在网页上，无经确定，就直接跳转到Apple Store下载你的应用，那是不可能的。想让网页与本地应有之间有交互，iOS 8带来更多的便捷性。

1. Safari插件
2. 共用认证（web和本地应用之间能共用安全凭证，无需再重新登录）。

新的webview

这次iOS 8更新，最令人激动的消息就是混合应用与 Mac 上共同了一套API，意味着iOS 上也能有更多的功能，Mac 和 iOS的通信交互，还有：

1. 支持JavaScript与本地应用之间通过postMessage的交互
2. New classes configure the Web View similar to the power we have on Android’s Web View.（这段谁能翻译下）
3. 更强大的 Nitro引擎，相比前代4x速度的js执行速度。

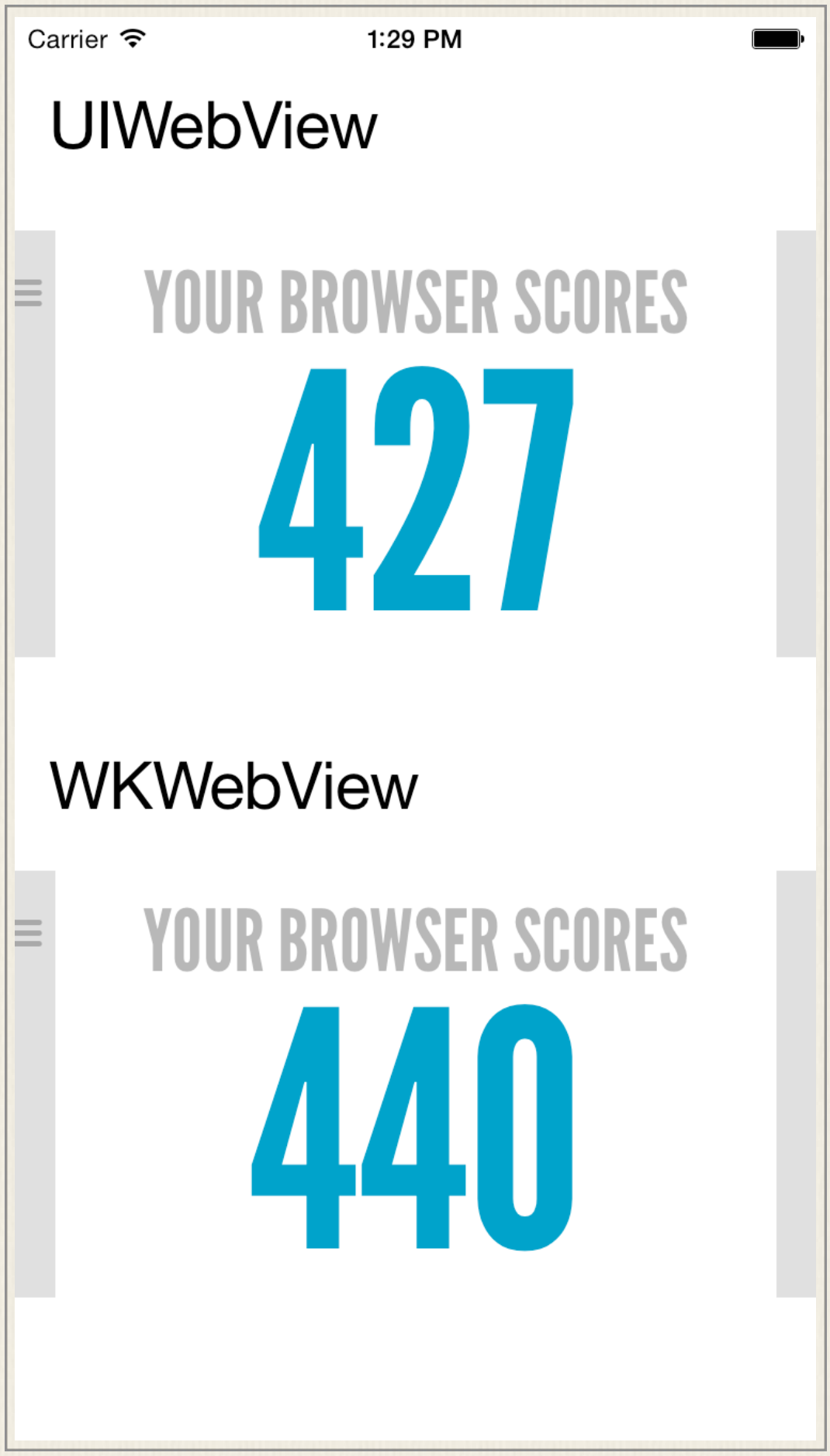
新的 webview(WKWebkit)是新框架(WebKit.framework)的一部分，与老的UIWebView并不是完全兼容。但是老的webviwe依旧保留，所以老的webapp还是会使用老的webview。

目前的GM版本（公开版也是一样的）依旧存在本地文件无法上传的bug，对于一些混合应用（例如Cordova PhoneGap）来说，这（多webvi-we）算是个好消息。

这就意味着，目前 iOS 8，拥有4个web引擎，当然，也意味着兼容性和bug都是有差异的。 1. Safari

2. Web.app (使用full-screen 桌面应用)
3. UIWebView (老)
4. WKWebView（新）

你可以在 HTML5Test.com这里测试下你webapp的性能。



对于那种包壳应用和webviewe应用来说，这个变化是十分重要的。比如说iOS上的Chrome和Facebook本地应用中的网页应用（我们猎豹的电池医生、手机猎豹也大量应用webview）。根据一份报告，11.5%的iOS流量是来自基于webview的应用。

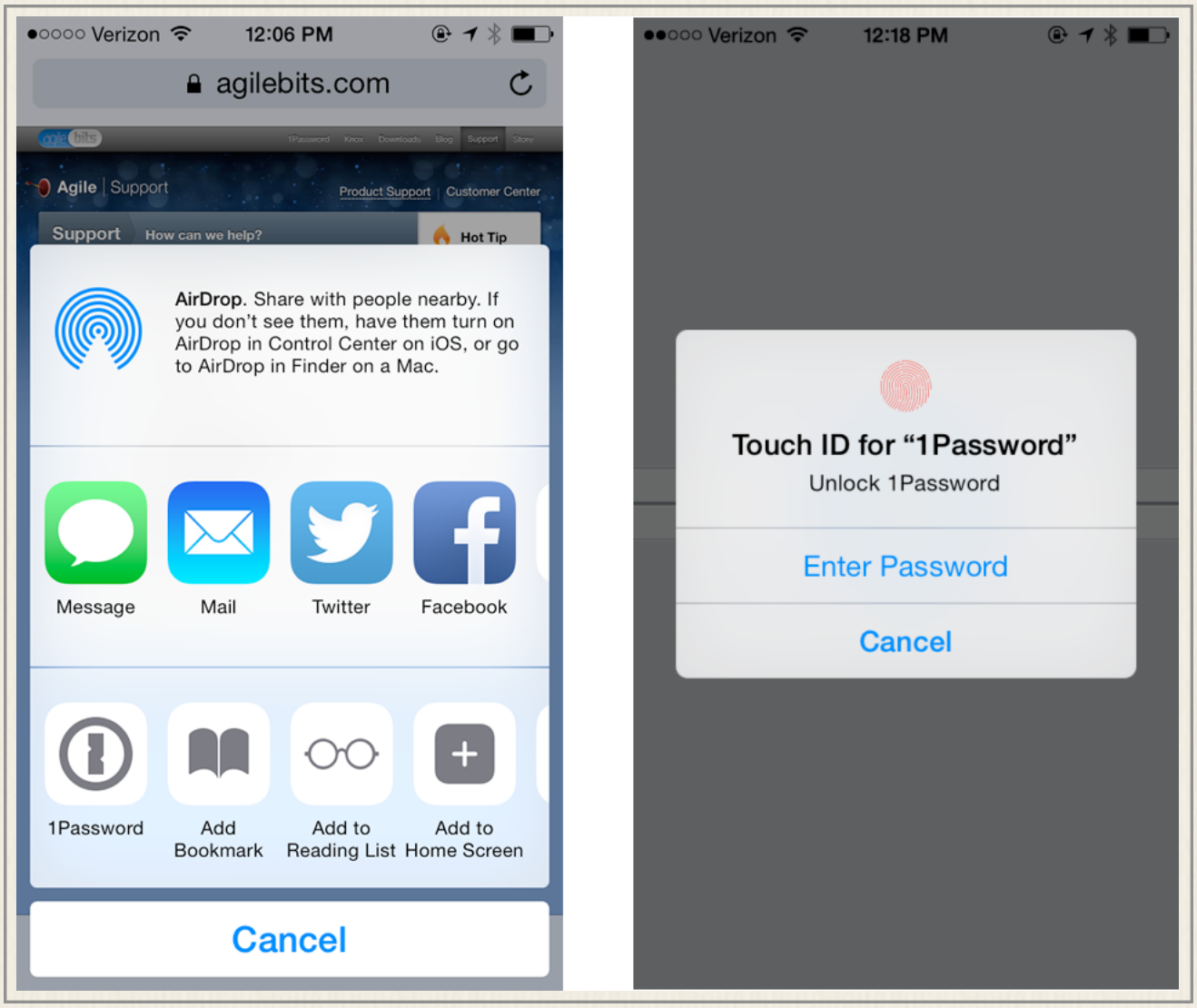
Safari 插件

iOS 8上的Safar 是第一个支持插件和拓展的系统预装浏览器（Firefox OS也许也算是一个）。从iOS 8开始，本地应用可以拓展到与 Safari 交互，主要通过两种方式：分享(Share extension)和动作(Actions)。Action 可以与DOM交互，意味着可以本地应用可以直接修改DOM元素。

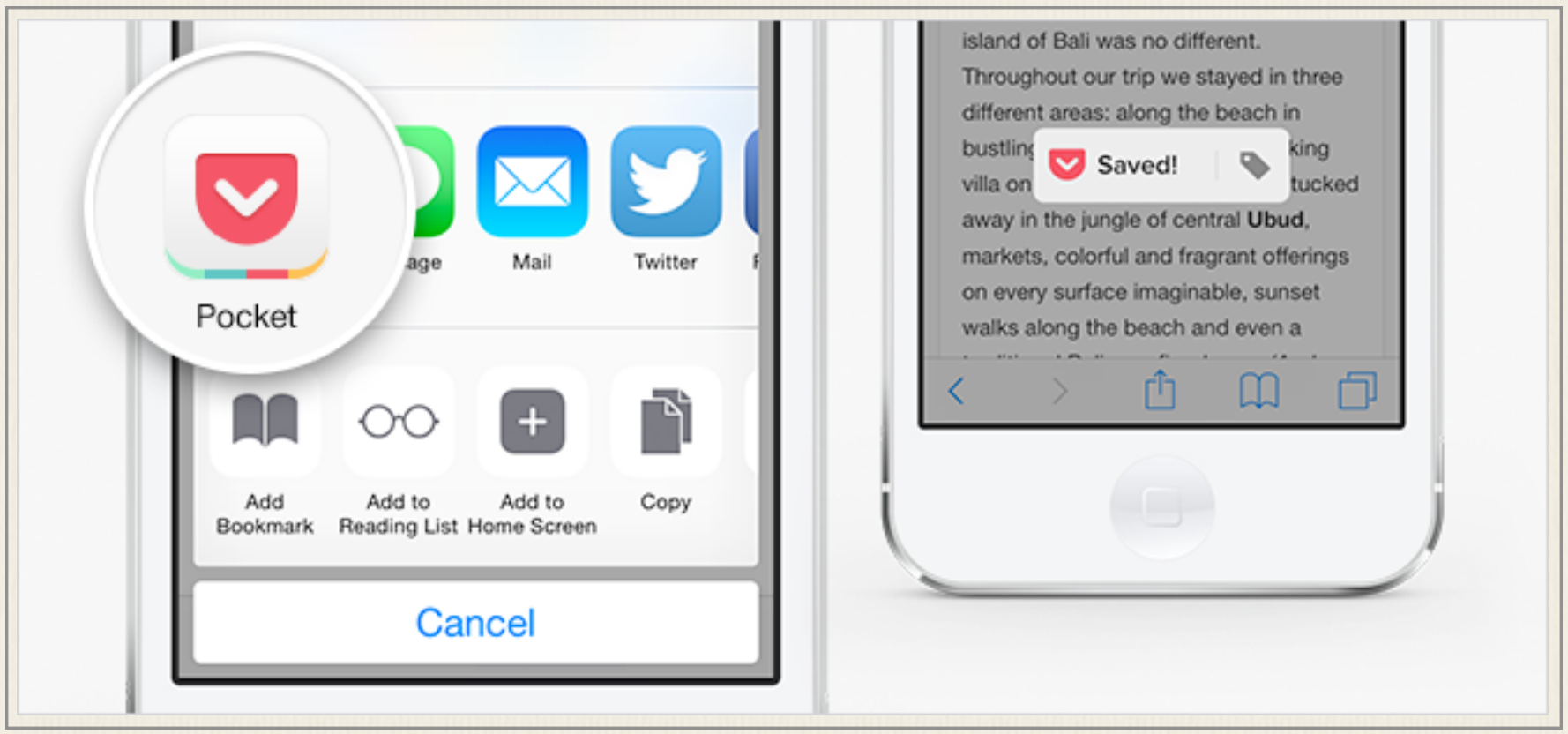
所有的插件都需要用户主动选择来触发（需要点击分享中不同的按钮来触发），暂时还没有可以自动运行的插件。

旧版 Safari 的分享 菜单同样被基于 JavaScript 的插件所替代。比如说添加到桌面如今就是一段 Safari 中的 js 代码。

除了苹果在WWDC上演示的功能，Safari的这个改进意味着浏览网页会有极大的体验改善。比如说你可以在Safari直接调用 1Password或者Last-Password 保存的帐号密码登录，如果你的iPhone 有 Touch ID指纹识别，你甚至可以直接指纹验证登录。



Pocket.com 已经声明即将推出针对 iOS 8的插件。



前端和设计师的福利

这次safari新增了很多html5，css3的支持

- CSS Shapes
- CSS object-fit
- CSS Background Blend modes
- word-spacing
- CSS Compositing and Blending
- Subpixel layout 支持小数点
- Animated PNG supported APNG格式图片
- Parallax effects and Pull-to-refresh supported （与Scroll事件相关）
- SVG Fragments Identifiers (for SVG Sprites)
- Image Source Set support
- HTML Template support

Animated PNG

APNG格式是PNG位图动画的拓展，但未获得PNG组织官方的认可，与GIF有点类似（只不过GIF是基于JPG的），这意味着我们可以制作32位全彩半透明的动态图。

滚动时差与下拉刷新

如果你做过移动端的项目，你一定知道iOS7以及以前都不支持scrooll事件，iOS 8 终于支持滚动事件，这下大家终于可以在iOS上用到视觉滚差相关的js和css了，但是不保证完全适配。

这个支持让我们可以做出下拉刷新和无限下拉下载的效果。

小数点单位

Safari现在CSS单位从整数转成了浮点数。这意味着CSS对象模型中诸如offsetTop和ClientWidth可能会取得小数值，之前老的iOS都会返回整数值。

与此同时，这也意味着你可以用半个像素单位了。

```
div {
  border-width: 0.5px;
}
```

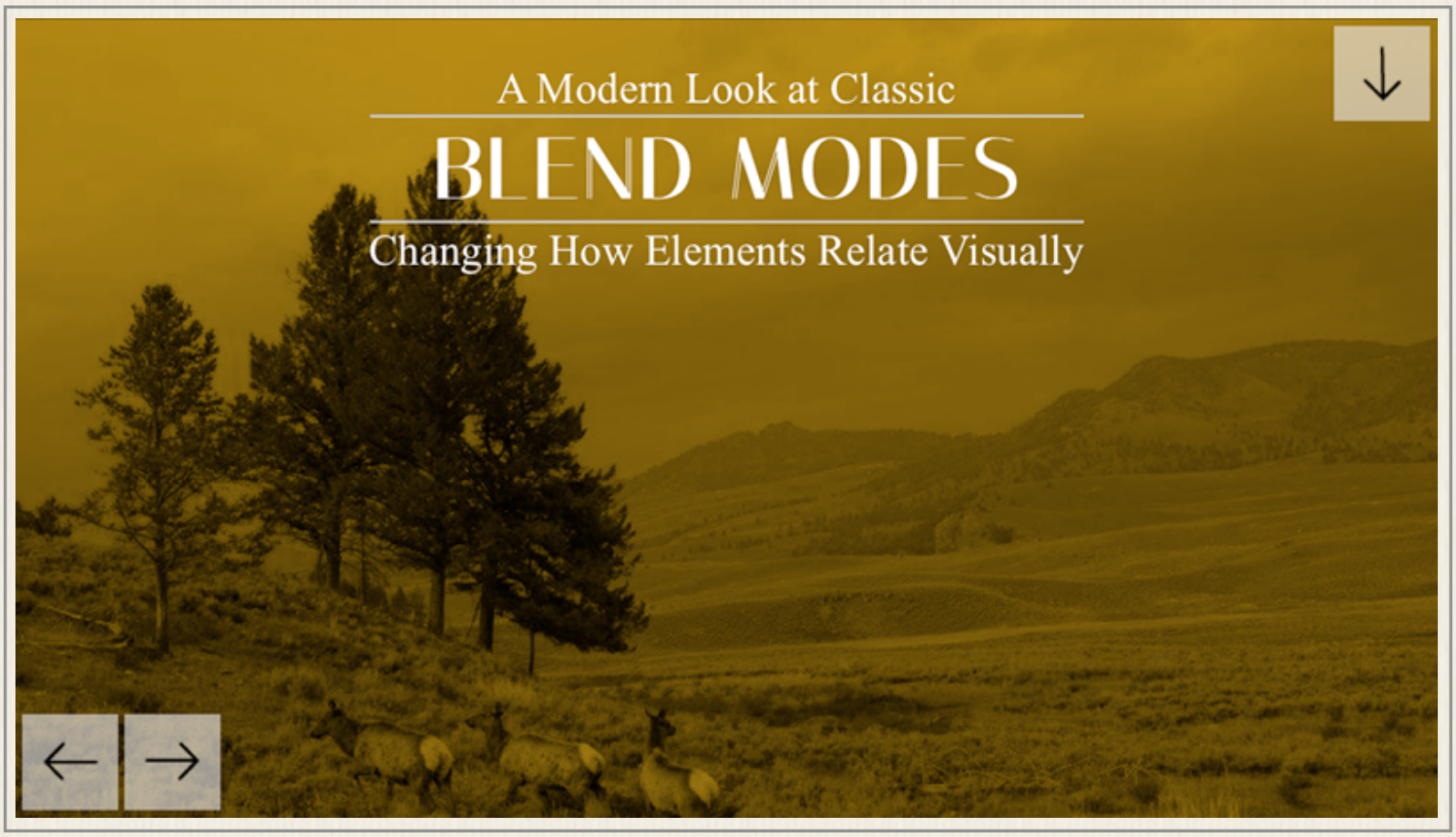
SVG 片段标识

SVG没怎么玩过，这个属性也不懂，大家先看英文吧。

Fragment identifiers from SVG is a method to link to one specific fragment or portion of an SVG instead of the root element. This feature allow us to sprite SVG images in one file taking advantage of one HTTP request and caching. Similar to CSS Sprites, but with SVG images instead and with ids instead of positions.

CSS 合成和变形

利用mix-blend-mode这个新属性支持我们把不同形状的不同元素合成到一张图片，这是Adobe在HTML5方面新出的一个功能。在他们的网站上你能看到演示的DEMO和文档。



从Adobe 官网的例子演示来看，似乎变形是生效了，合成并没有生效（我Mac Chrome也没生效）。

CSS 形状

CSS 形状也是Adobe 新出的一个特性。对于这个特性，推荐下W3Cplus上的《CSS Shapes 101》。个人还是很看好这个特性的推广和应用的，能给我们的网页设计带来更多的可能性。

Image Source Set

有了解响应式图片的开发者对这个应该不陌生，随着高清屏的普及，针对不同的分辨率做适配是一个需要注意的问题，苹果iOS 8 支持Image Source Set Spec，意味着可以在<img>标签中使用新的属性。

<img src="lores.png" srcset="hires.png 2x, superhires.png 3x">

在上面这段代码的例子中，iPhone 6 Plus 的像素比是3x，它会加载superhires.png这张图片，而iPhone 5s,iPhone 6 则会加载hires.png这种图，其余的则加载lores.png。但是遗憾的是，iOS 暂时还不支持<picture>元素。

HTML模板

<template>对于webapp来说十分有用，<template>中可以包含一段css或者js（不会被浏览器解析），实际应用中，我们可以利用<template>中的代码创建一个新的node。

视频增强

这次更新对视频播放相关的增强了许多，iOS8开始支持全屏播放,Meta Data Api和CSS layering(话说大家知道HTML5 Video元素是可以通过CSS来控制吧？)

video元素的全屏播放

Safari不支持全屏播放API，即使是iOS8也不例外。但是能通过一个特殊的方法解决这个问题，在<video>元素中增加一段js。

<input type="button" value="Go Full screen" onclick='document.querySelector("video").webkitEnterFullScreen()'>

视频 **Metadata API**

Safari开始支持<video>的preload="metadata"，可以让浏览器触发loadedmetadata事件方便开发者控制。

CSS 分层

我可以通过css控制其他元素放在<video>之前。

iOS 8 JavaScript相关

- 部分支持ECMAScript 6,包括Promises, Iterators, Maps, For-of, Weak Maps等等。
- 后台运行：JavaScript会在后台继续运行（哪怕已经切换了窗口甚至Safari切换到后台，但是计时器 times 下降到1s的频率）。
- 支持scroll事件：不多说了，上面介绍过。
- Unprefixed Page Visibility AP： 我没搞懂这个是什么（哪位知道求指导）

- 移除window.doNotTrack
- 支持window.currentScript

9月18更新:

有开发者发现：iPhone 5上的click事件300ms延迟已经取消了（只是Safari），但是其他设备上的Safari和 WebViews 上还有，延迟依旧在 iPod touch, iPads 和iPhone 5s上。

Bug和存在的问题

说了iOS8和iPhone 的更新，再来说说目前发现的 Bug。

- 不支持文件上传！！！所有的文件上传都失效了，你能选择或者拍张照片，但是js不能获得任何数据。HTML或者XMLHttpRequest的POST请求也不行。但是这个问题但是在桌面启动的app没发生。
- WKWebKit：在新的引擎不能读取本地文件是个大问题，所以对于混合app来说，还得使用老的UIWebView。
- window.prompt可能会导致Safari崩溃
- 附件:语音在input和labels输入时，不管用。(当输入的时候，label不再发音)
- 桌面app的iframes中,touch事件没有被监测到。
- 手机休眠后，桌面app中的Timers和requestAnimationFrame回调没有执行。

译者言

iOS 8给web前端带来的变化还是很令人欣喜的，更强大的性能，更强大的浏览器和内核，更开放的接口，可以预见在针对iOS的web应用会有一个更大的用武之地。

文章很长，大量技术词汇，翻译了半天，若译文有错误或者不妥之处，欢迎指正和提供建议。

更新

2014年9月26日:iOS 8.02 推送更新后，解决了Safari 无法上传文件的Bug。

原文链接：http://luolei.org/safari-ios8-iphone6-web-developers-designers-chinese/

iOS 8人机界面指南（一）：UI设计基础

译者：糖箔糊

- 1.1 为iOS而设计（Designing for iOS）
- 1.1.1 以内容为核心（Defer to Content）
- 1.1.2 保证清晰度（Provide Clarity）
- 1.1.3 用深度来体现层次（Use Depth to Communicate）
- 1.2 iOS应用解析（iOS App Anatomy）
- 1.3 适应性和布局（Adaptivity and Layout）
- 1.3.1 为自适应而开发（Build In Adaptivity）
- 1.3.2 在不同环境提供良好体验（Provide a Great Experience in Each Environment）
- 1.3.3 使用布局来沟通（Use Layout to Communicate）
- 1.4 起始与停止（Starting and Stopping）
- 1.4.1 即时启动（Start Instantly）
- 1.4.2 时刻准备好停止（Always Be Prepared to Stop）
- 1.5 导航（Navigation）
- 1.6 模态情境（Modal Contexts）
- 1.7 交互性和反馈（Interactivity and Feedback）
- 1.7.1 用户知道标准手势（Users Know the Standard Gestures）
- 1.7.2 可交互元素吸引用户点击（Interactive Elements Invite Touch）

- 1.7.3 反馈有助于理解（Feedback Aids Understanding）
- 1.7.4 输入信息的方式要简单（Inputting Information Should Be Easy）
- 1.8 动画（Animation）
- 1.9 品牌推广（Branding）
- 1.10 颜色与字体（Color and Typography）
- 1.10.1 色彩有助于增进沟通（Color Enhances Communication）
- 1.10.2 文字应该清晰易读（Text Should Always Be Legible）
- 1.11 图标和图形（Icons and Graphics）
- 1.11.1 应用图标（The App Icon）
- 1.11.2 栏图标（Bar Icons）
- 1.11.3 图形（Graphics）
- 1.12 术语和措辞（Terminology and Wording）
- 1.13 与iOS的整合（Integrating with iOS）
- 1.13.1 正确使用标准UI元素（Use Standard UI Elements Correctly）
- 1.13.2 弱化文件和文档处理（Downplay File and Document Handing）
- 1.13.3 必要时提供可配置选项（Be Configurable If Necessary）
- 1.13.4 充分利用iOS技术（Take Advantage of iOS Technologies）

1.1 为iOS而设计（Designing for iOS）

iOS 的革新关键词如下：

- 遵从：UI能够更好地帮助用户理解内容并与之互动，但却不会分散用户对内容本身的注意力。
- 清晰：各种大小的文字应该易读，图标应该醒目，去除多余的修饰，突出重点，很好地突显了设计理念。
- 深度：视觉的层次和生动的交互动作会赋予UI新的活力，不但帮助用户更好理解新UI的操作并让用户在使用过程中感到惊喜。

无论你是重新设计一个现有的应用或是重新开发一个，请尝试下列方法：

- 首先，去除了UI元素让应用的核心功能呈现得更加直接并强调其相关性。
- 其次，直接使用iOS的系统主题让其成为应用的UI，这样能给用户统一的视觉感受。
- 最后，保证你设计的UI可以适应各种设备和不同操作模式，这样用户可以在不同场景下舒适地享用你的应用。

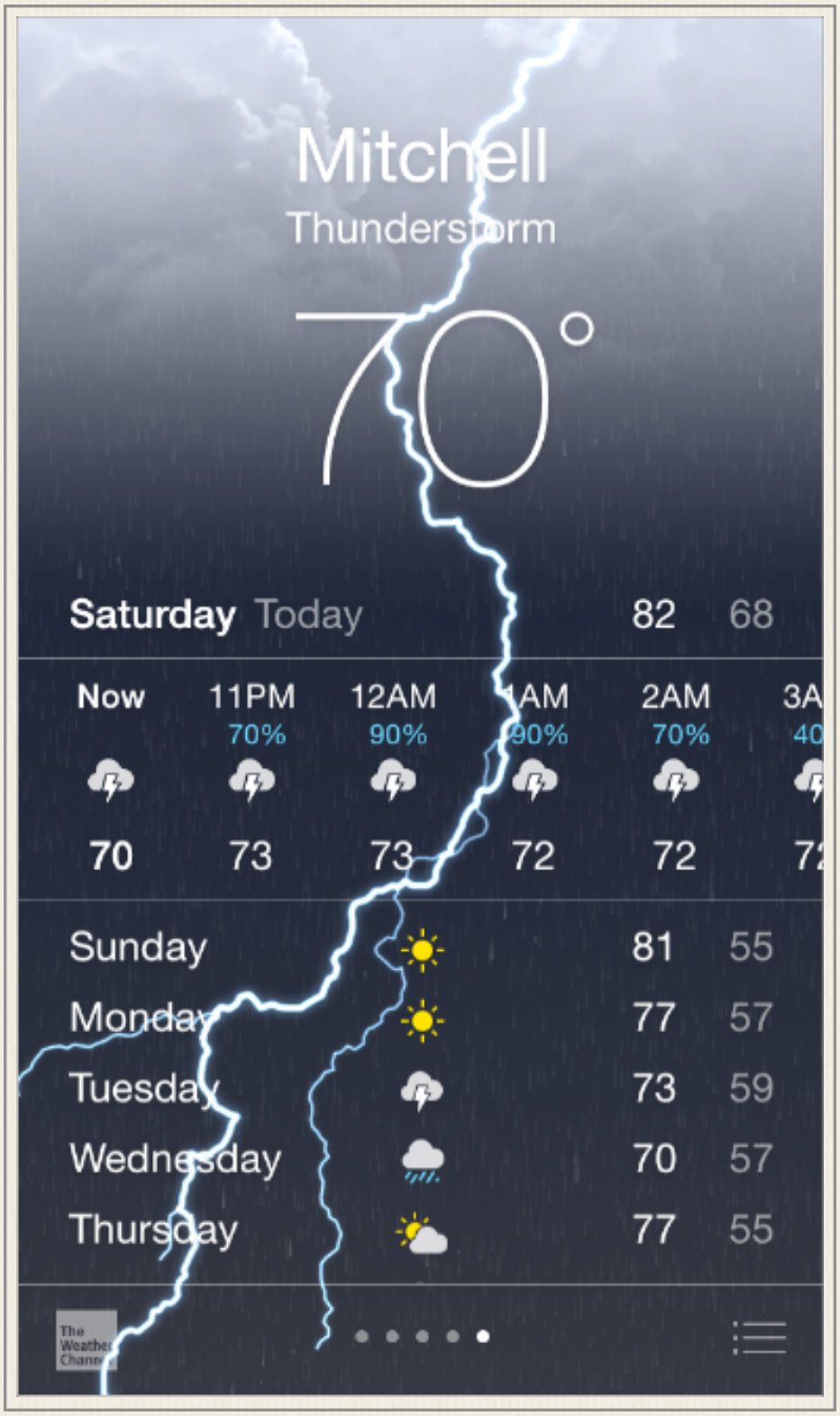
在整个设计过程中，时刻准备着推翻先例，假设问题，并以重点内容和功能（为目标）来驱动每个细节设计。

1.1.1 以内容为核心（Defer to Content）

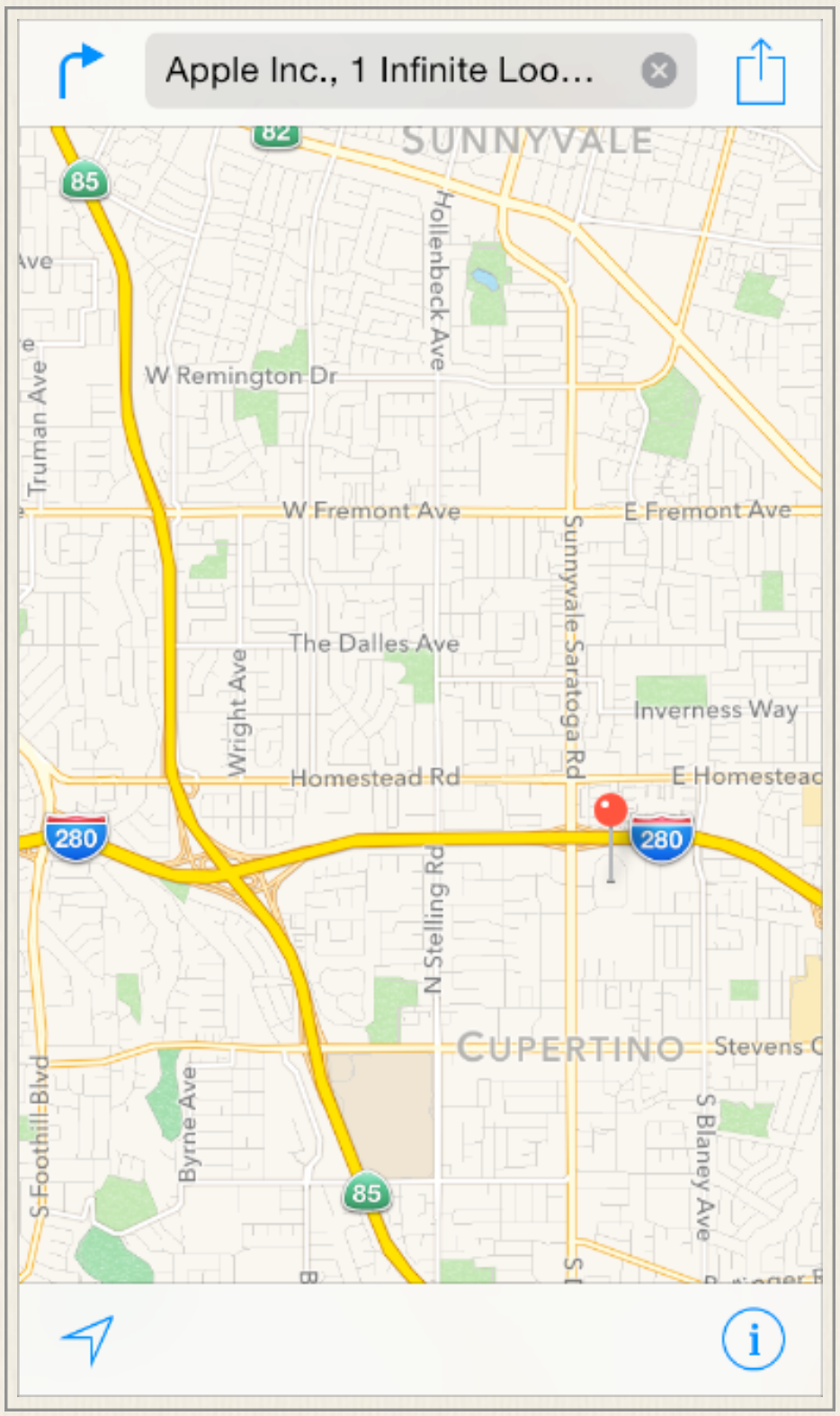
虽然明快美观的UI和流畅的动态效果是iOS体验的亮点，但内容始终是iOS的核心。

这里有一些方法，以确保你的设计能够提升你的应用功能体验并关注内容本身。

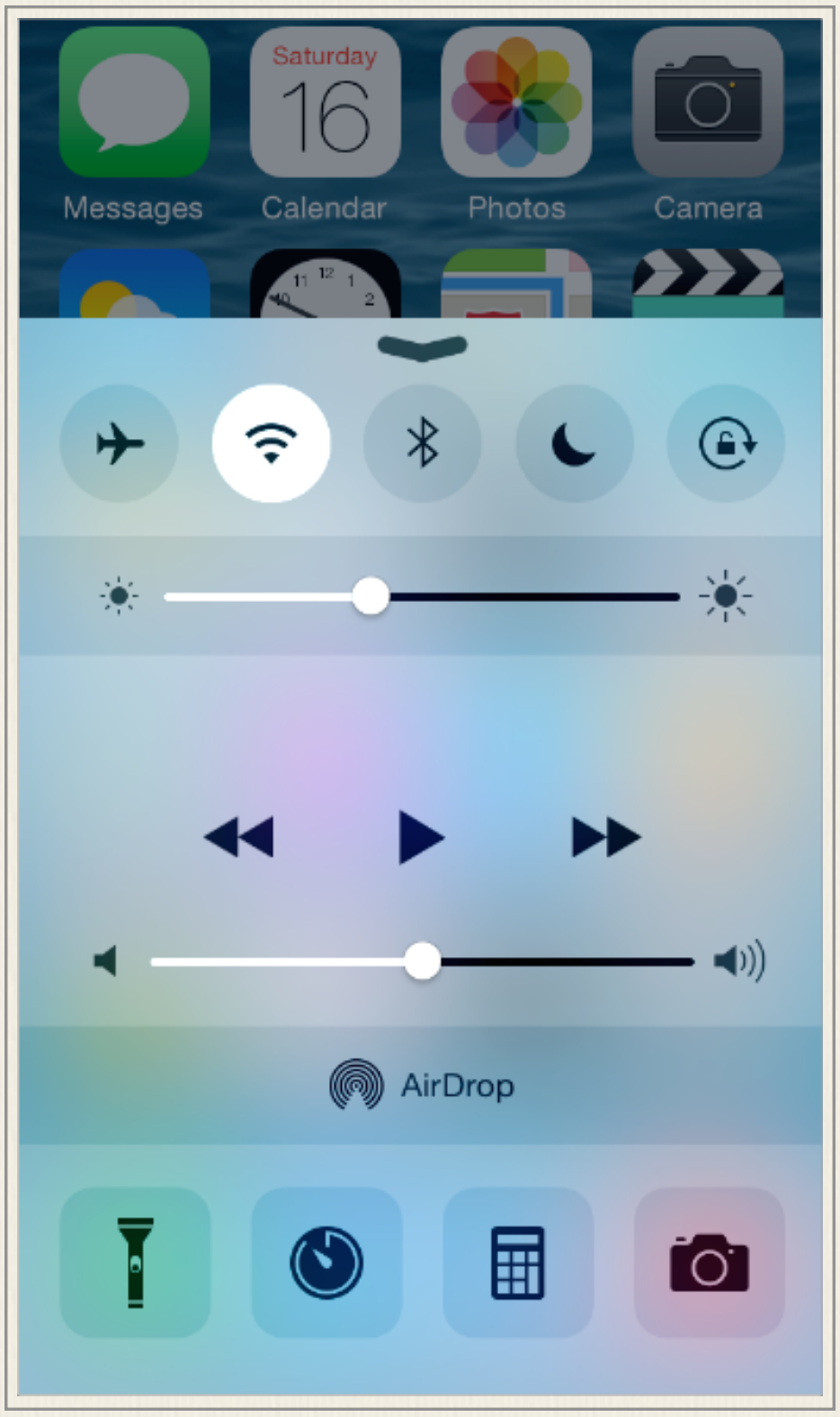
充分利用整个屏幕。天气应用是最好的例子：漂亮的天气图片充满全屏，呈现用户所在地当前天气情况这最重要的信息，同时也留出空间呈现了每个时段的气温数据。


Mitchell
Thunderstorm
70°
Saturday Today 82 68
Now 11PM 12AM 1AM 2AM 3A
70% 90% 90% 70% 40
70 73 73 72 72 7
Sunday 81 55
Monday 77 57
Tuesday 73 59
Wednesday 70 57
Thursday 77 55

尽量减少视觉修饰和拟物化设计的使用。UI面板、渐变和阴影有时会让UI元素显得很厚重，致使抢了内容的风头。应该以内容为核心，让UI成为内容的支撑。



尝试使用半透明底板。半透明的控件——比如控制中心——关联了使用场景，帮助用户看到更多可用的内容，并可以起到短暂的提示作用。在iOS中，透明的控件只让它遮挡住的地方变得模糊——看上去像蒙着一层米纸一样——它并没有遮挡屏幕剩余的部分。

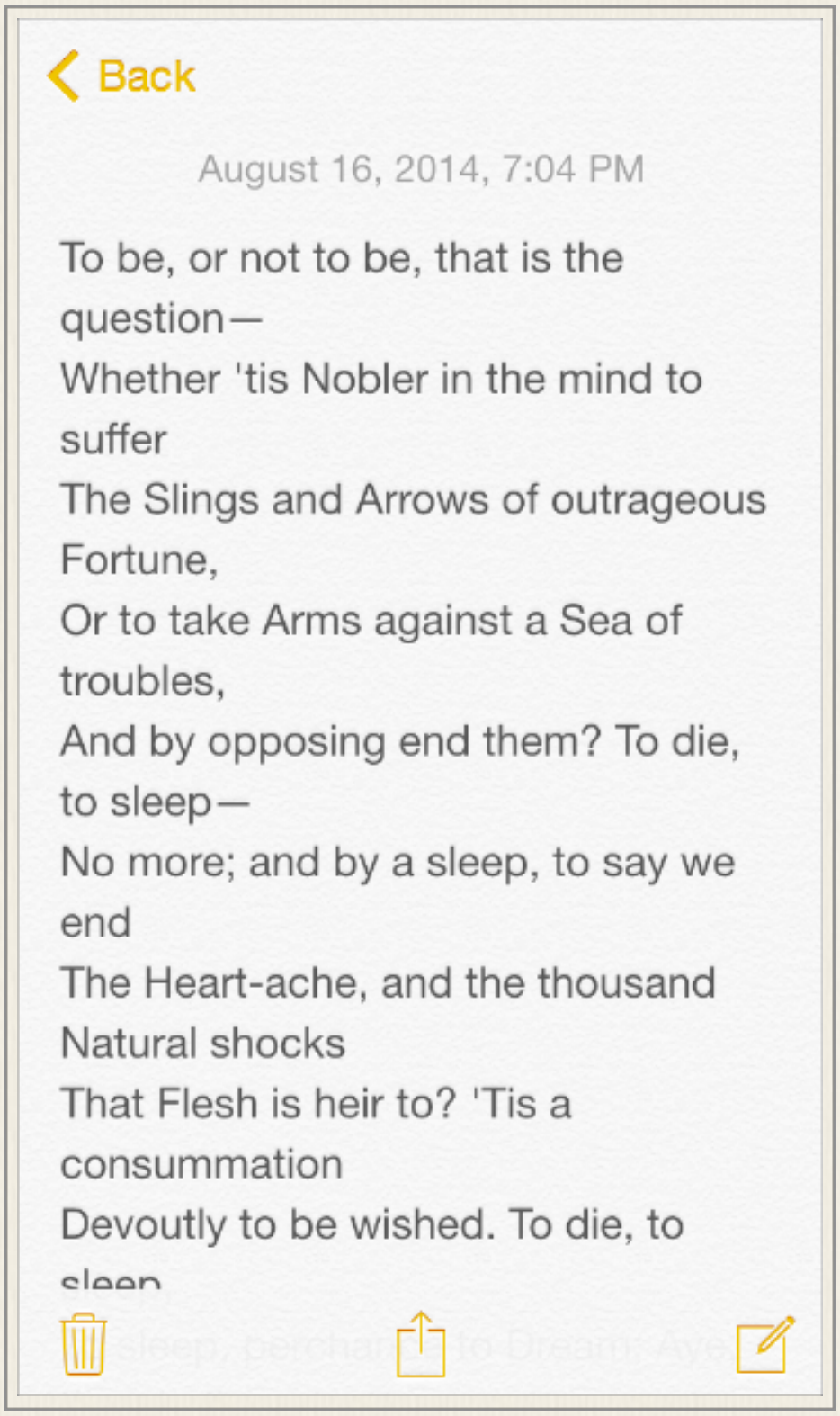


1.1.2 保证清晰度（Provide Clarity）

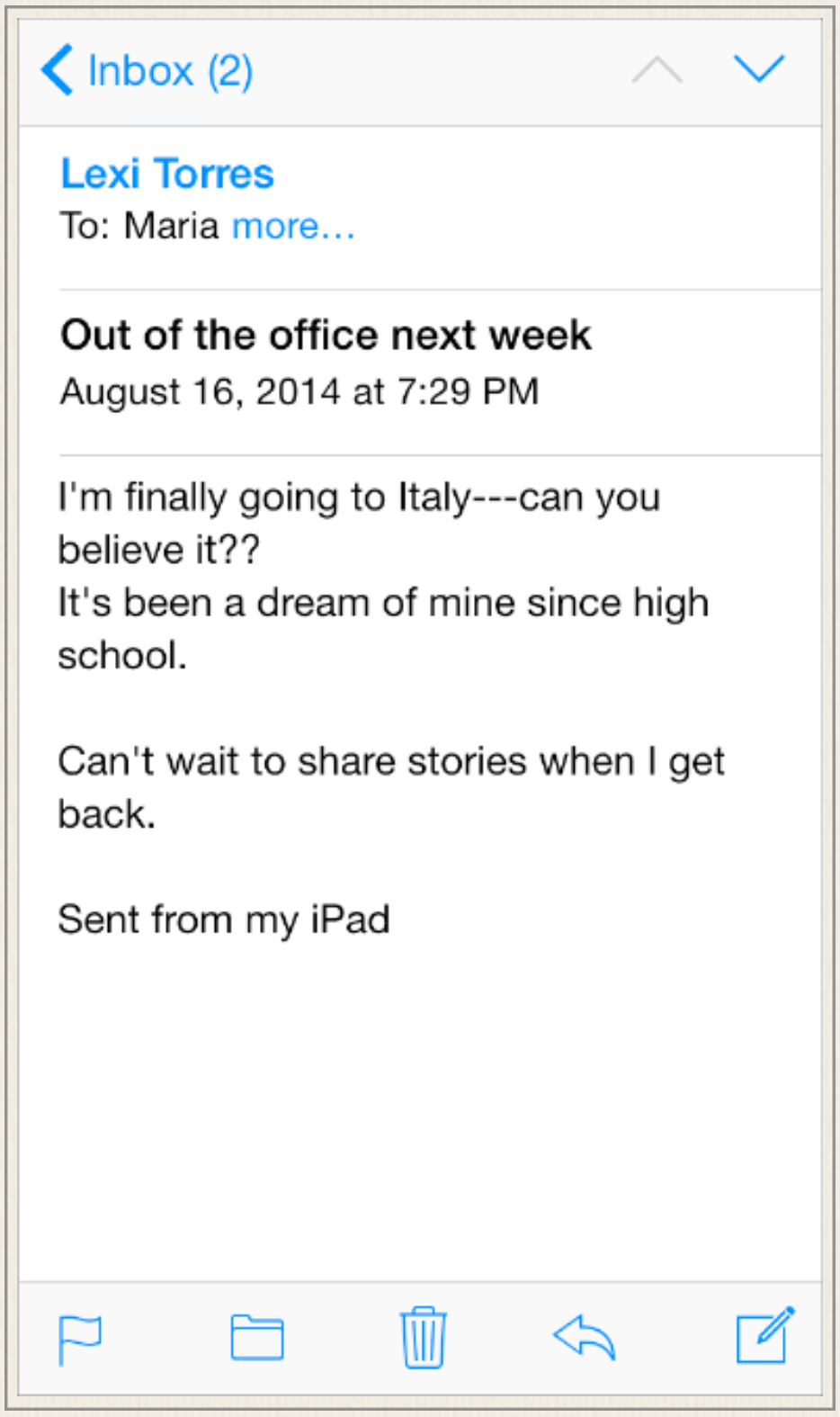
保证清晰度是另一个方法，以确保你的应用中内容始终是核心。以下是几种方法，让最重要的内容和功能清晰，易于交互。

使用大量留白。留白让重要内容和功能显得更加醒目。此外，留白可以传达一种平静和安宁的视觉感受，它可以使一个应用看起来更加聚焦和高效。

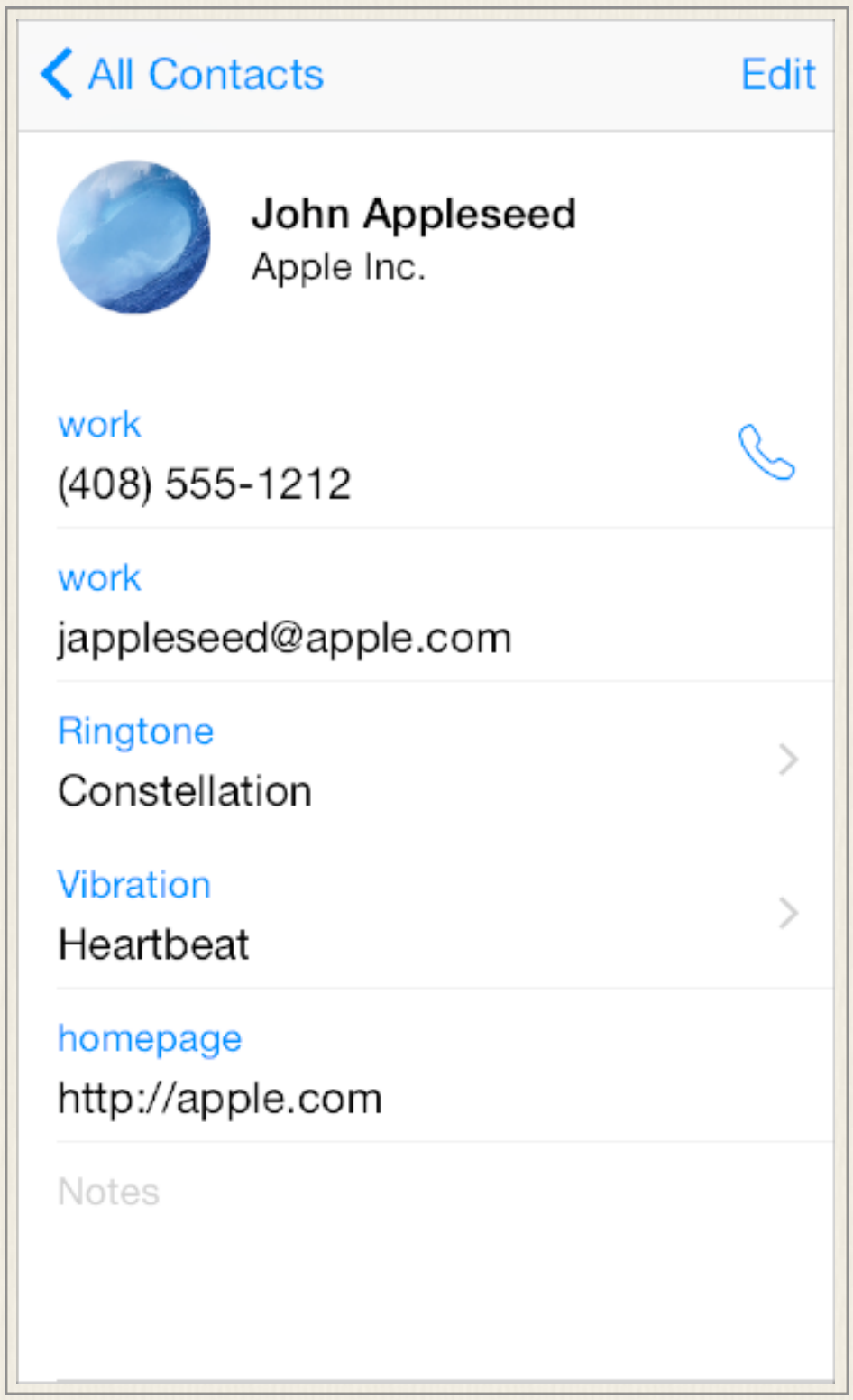
让颜色简化UI。一个主题色——比如在记事本中使用的黄色——让重要区域更加醒目并巧妙地表示交互性。这同时也给了一个应用一个统一的视觉主题。内置应用使用家族化的系统颜色，无论在深色和浅色背景上看起来都干净，纯粹。



通过使用系统字体确保易读性。iOS的系统字体自动调整行间距和行的高度，使阅读时文本清晰易读，无论何种大小的字号都表现良好。无论你是使用系统或是自定义字体，务必使用动态型，这样你的应用可以在用户选择不同字号时做出应对。



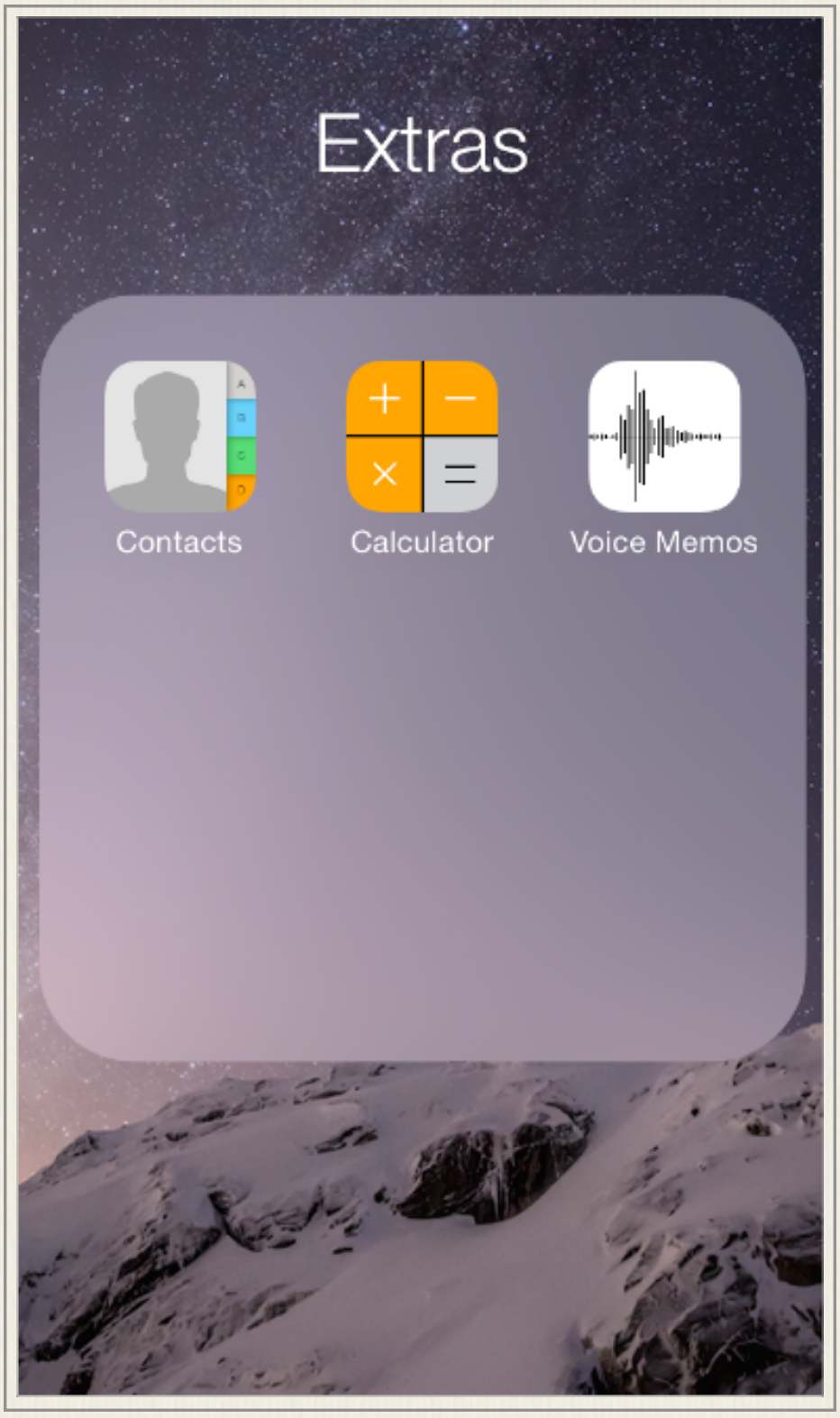
使用无边框的按钮。默认情况下，所有bar上的按钮都是无边框的。在内容区域，无边框按钮以文案、颜色以及操作指引标题来表明按钮功能。当按钮被激活时，该按钮呈现高亮的浅色状态来作为操作响应。



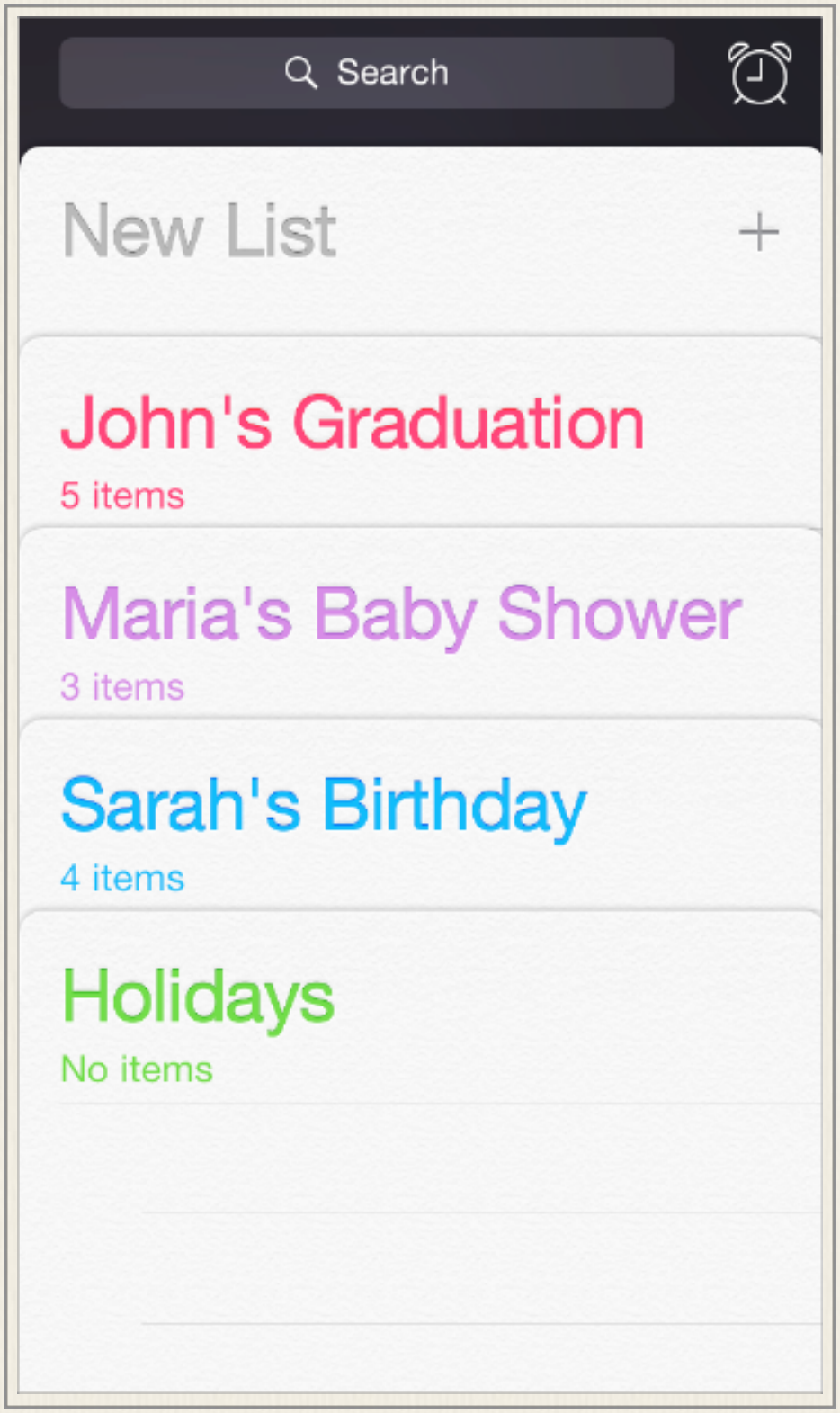
1.1.3 用深度来体现层次（Use Depth to Communicate）

iOS经常在不同的层级上展现内容，用以表达分组和位置，并帮助用户了解在屏幕上的对象之间的关系。

通过使用一个在主屏幕上方的半透明背景浮层来区分文件夹和其余部分的内容。

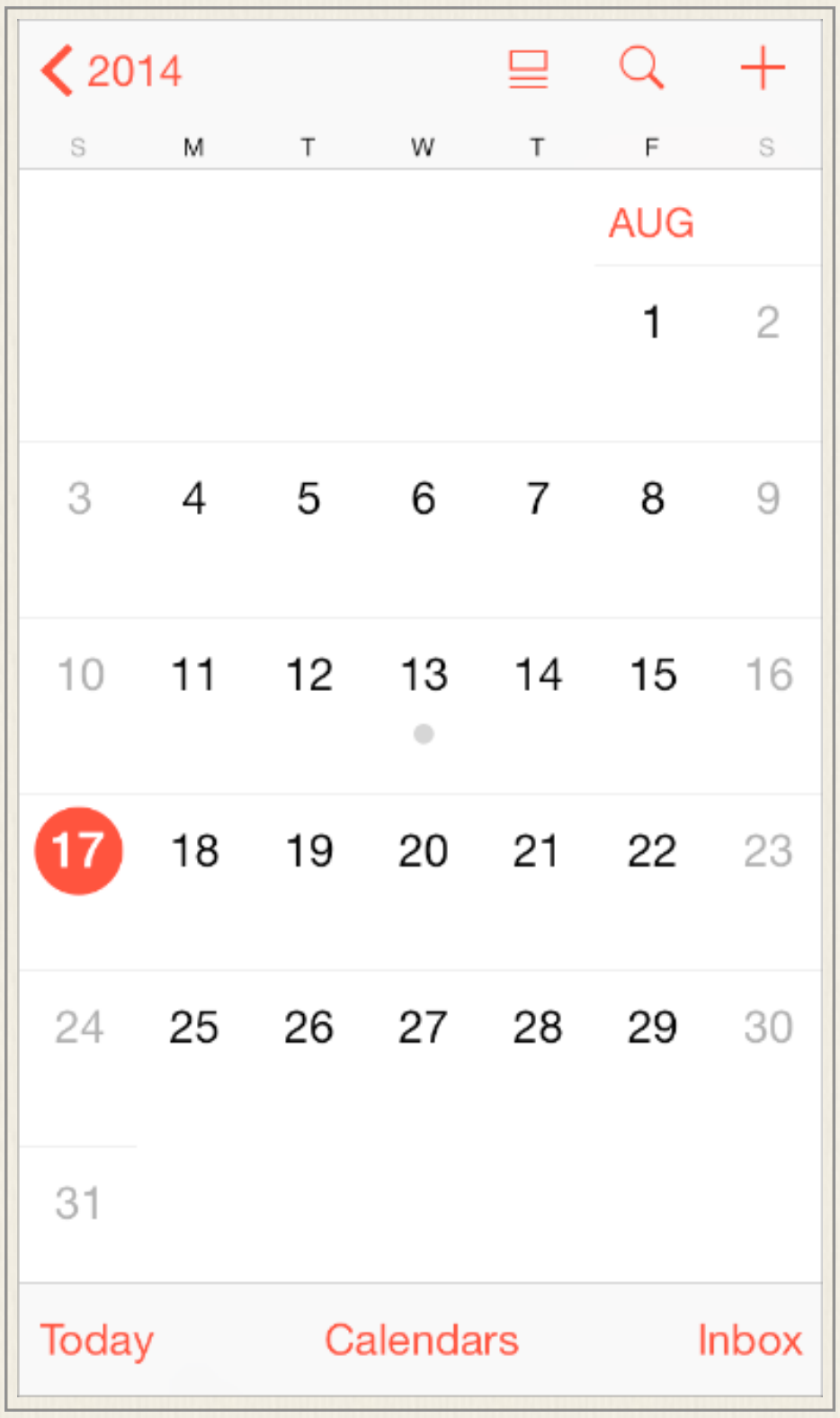


备忘录以不同的层级展示，如插图所示。用户在使用备忘录里的某个条目时，其他的条目被收起在屏幕下方（译者按：其实这个视觉提示使用起来很隐晦）。

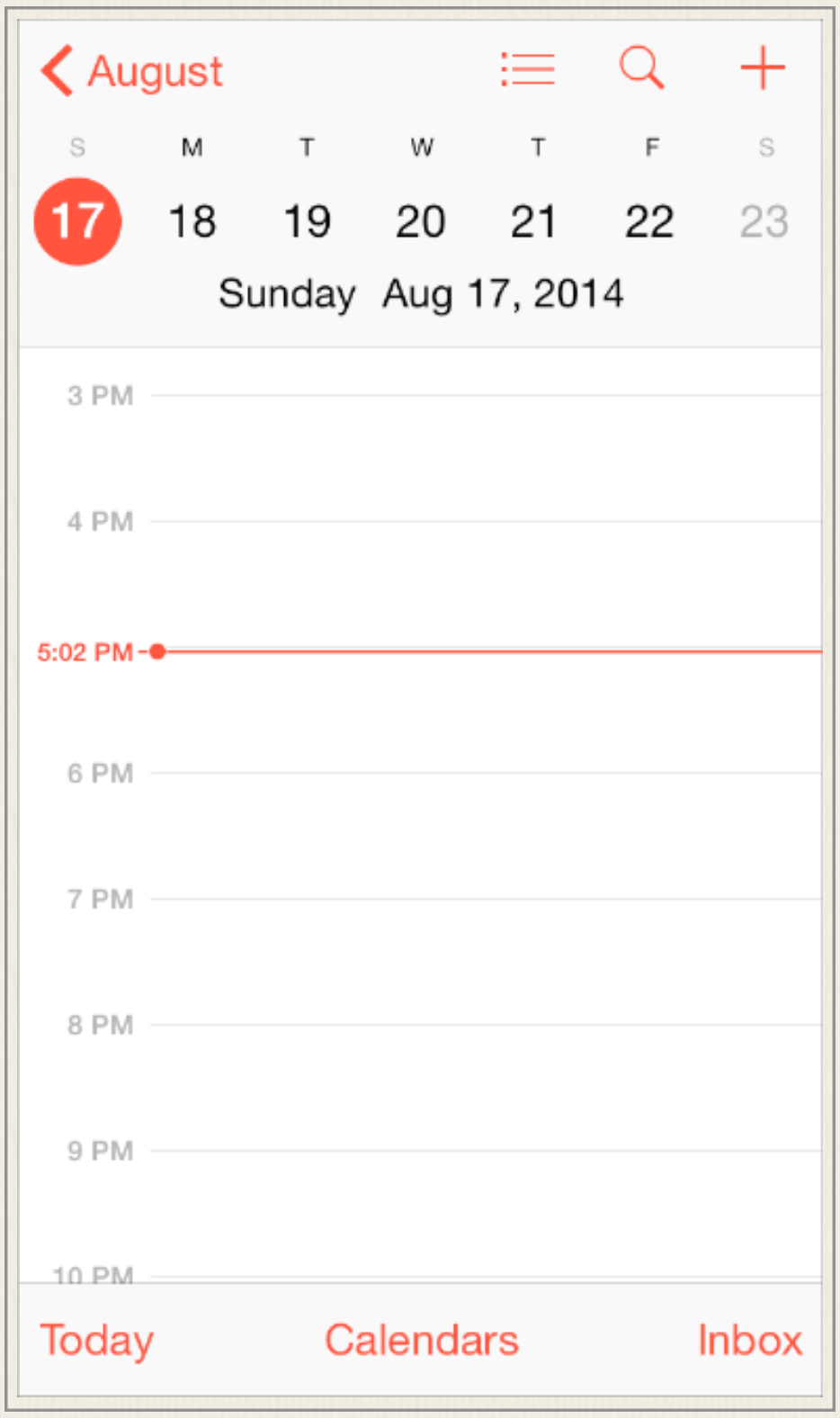


日历有较深的层级，当他们在翻阅年、月、日的时候，以及增强的交互动画给用户一种层级纵深感（循序切换的层次，从年到月到日）。在滚动年份视图时，用户可以即时看到今天的日期以及其他日历任务。当用户处于月份视图时，点击年份视图按钮，月份会缩小至年份视图中的所处位置。

今天的日期依然处于高亮状态，年份出现在返回按钮处，这样用户可以清楚地知道他们在哪儿，他们从哪里进来并且知道如何返回。

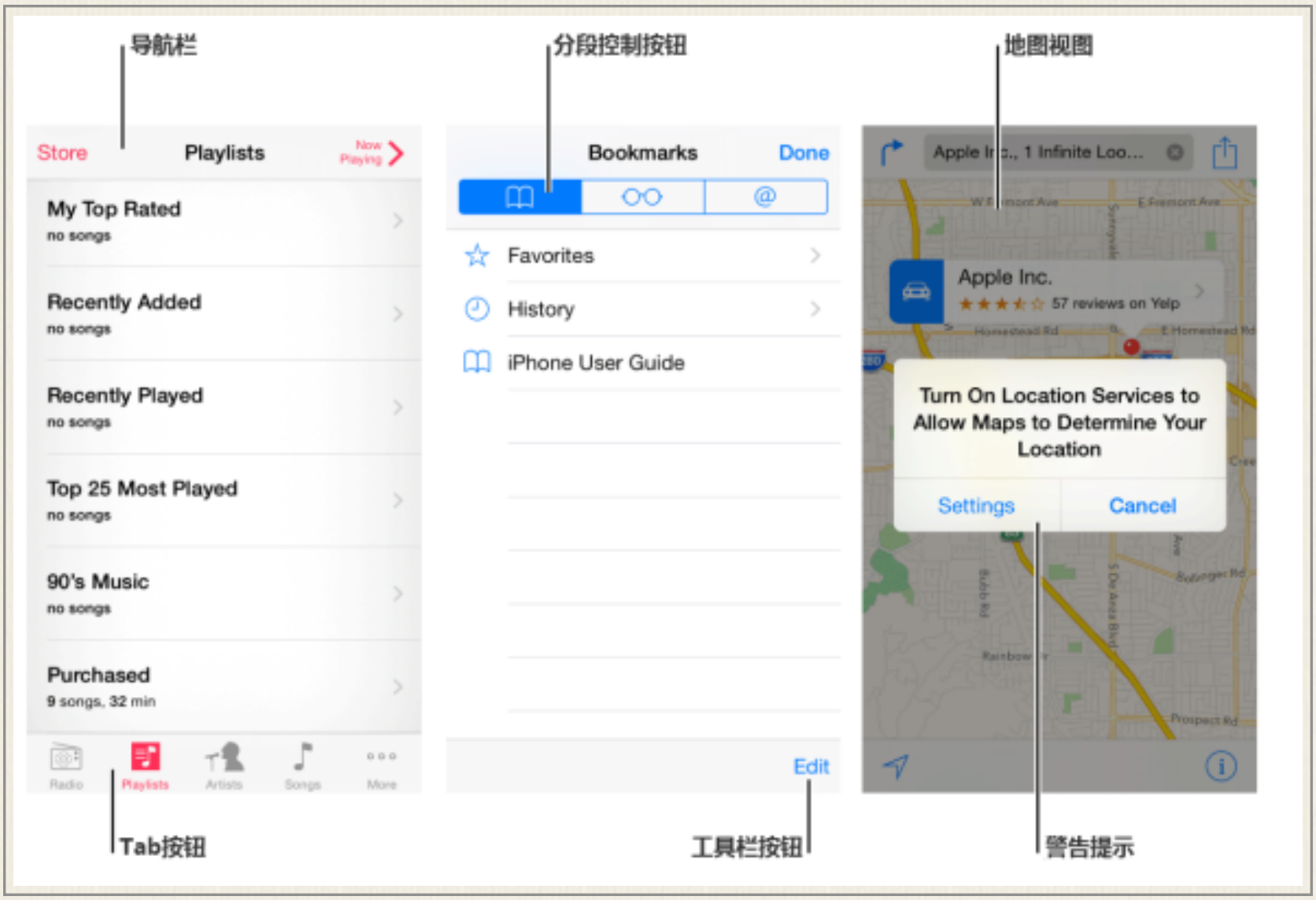


类似的过渡动画出现在用户选择一个日期时：月份视图从所选位置分开，将当前的周日期推向屏幕顶端并翻转出以小时为单位的当天时间视图。这些动画加强了日历上年月日之间的关系的感知度。



1.2 iOS应用解析（iOS App Anatomy）

几乎所有的iOS 应用都应用了UIKit framework中定义的组件。了解这些组件的名字，作用和构成能够帮助你设计应用过程中做出更好的决定。



UIKit提供的UI组件大致分成以下4种大类：

Bars：包含了导航信息，告诉用户他们所在的位置并包含了一些能帮助用户浏览或启动某些操作的控制按钮。

内容视图：包含了应用的主体内容以及某些操作行为，比如滚动、插入、删除、排序等等。

控制按钮：展示信息或者控制动作。

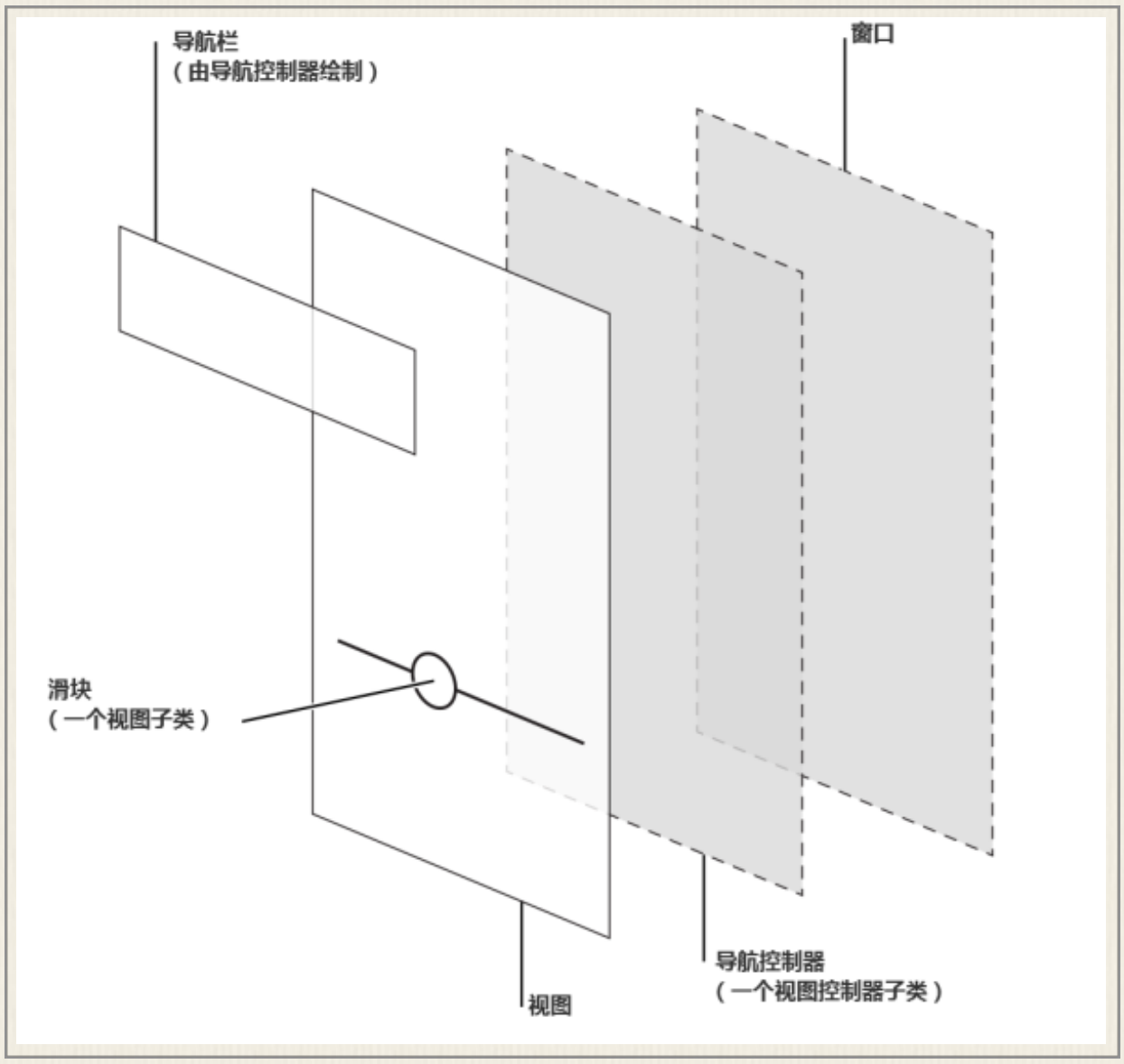
临时视图：短时间出现，给用户重要信息或者额外的选择或者其他功能。

除了定义UI组件，UIKit也定义对象实现的功能，例如手势识别、绘图、辅助功能和打印支持。

从编程的角度来说，UI组件被认为是不同类别的视图，因为他们从UIView得到继承。视图能绘制屏幕内容并且知道用户何时触摸了屏幕。视图的所有类型有：控件（比如按钮和滑块），内容视图（比如集合视图和表格视图），以及临时视图（如警告提示和动作菜单）。

要在应用中管理一组或者一系列的视图，通常需要使用一个视图控制器，它能协调视图的显示内容，实现与用户交互的功能并能在不同屏幕内容之间切换。比如，“设置”使用了一个导航控制器来展示其视图层级。

下面是一个例子，关于视图与视图控制器如何结合并呈现iOS 应用的UI。



虽然开发者认为真正起到作用的是视图和视图控制器，但一般用户感知到的iOS应用是不同屏幕内容的集合。从这个角度来看，在应用里，屏幕内容一般对应于一个独特的视觉状态或者模式。

注：一个iOS应用程序包含一个窗口。但是，不同于计算机程序中的窗口，iOS窗口没有可见的部分并且不能在屏幕上被移动到另一个位置。很多iOS应用程序只有一个窗口；可以支持外部显示设备器的应用程序可以有不止一个窗口。

在iOS Human Interface Guidelines中，screen这个词和大部分用户理解的一样。作为一个开发者，你也许需要读一下其他与UIscreen相关的章节，这样你可以更好的了解如何关联外部屏幕。

1.3 适应性和布局（Adaptivity and Layout）

1.3.1 为自适应而开发（Build In Adaptivity）

人们通常想随心所欲地使用自己喜欢的应用程序。在iOS 8及未来的版本中，你可以使用不同分辨率和自动布局来帮助你定义屏幕布局视图，视图控制器以及需要随显示环境改变的视图（显示环境display environment的概念指的是设备的整个屏幕或者其中一部分，比如一个跳出菜单区域或一个分视图控制器的主视图部分）。

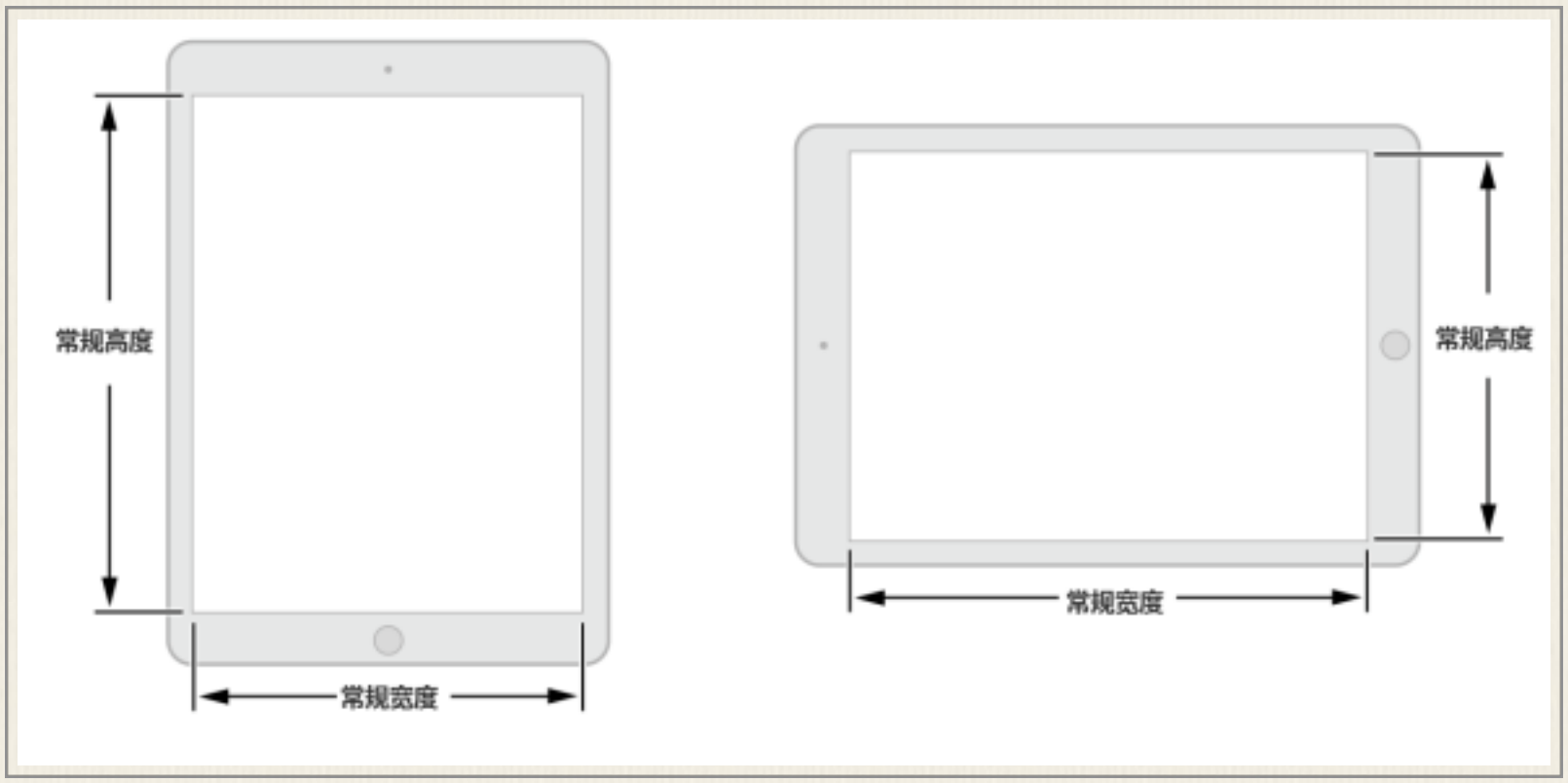
iOS定义了两个尺寸类别（size class），常规的（regular）和压缩的（compact）。常规尺寸有着较易拓展的空间，而压缩尺寸约束了空间的使用。想要定义一种显示环境， 你需要定义横纵尺寸类型。如你所想，iOS设备可以有横屏竖屏两种不同的使用模式。

iOS能随着显示环境和尺寸类别变化而自动生成不同布局。举个例子，当垂直尺寸从压缩变为常规时，导航栏和工具栏会自动变高。

当你靠尺寸类别来驱动布局变化时，你的应用在任何显示环境时都能显示得很好。关于如何在Interface Builder中更好的使用尺寸类别，你可以查阅Size Classes Design Help。

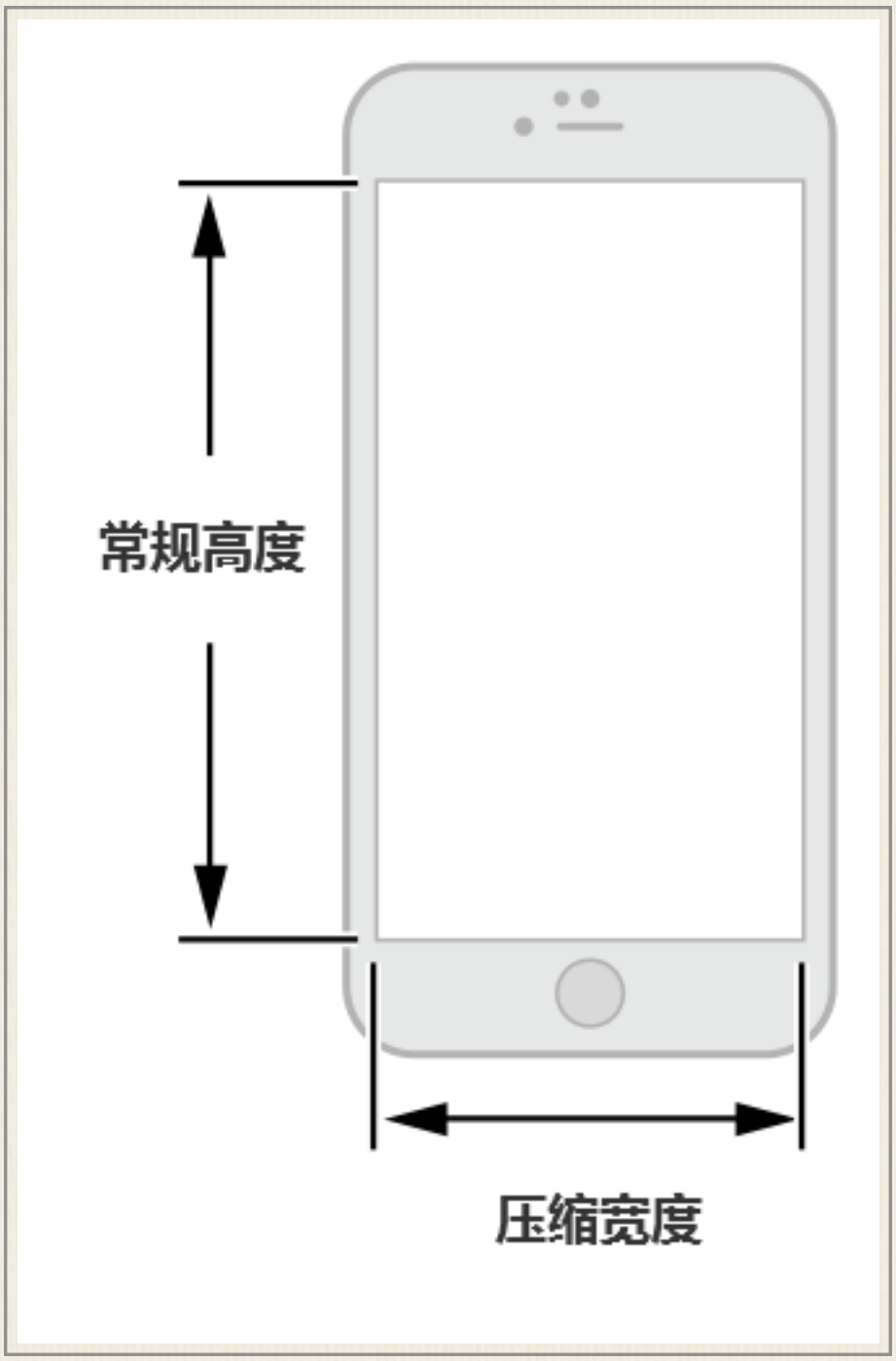
注：在一种尺寸类别中，持续使用Auto Layout进行小的布局调整，比如拉伸或压缩内容。

下面的实例可以帮助你理解尺寸类型是如何适配不同设备的显示环境。例如：iPad在长宽和横屏竖屏时都使用常规尺寸类型。换句话说，iPad显示环境一直处于垂直和水平的常规状态。

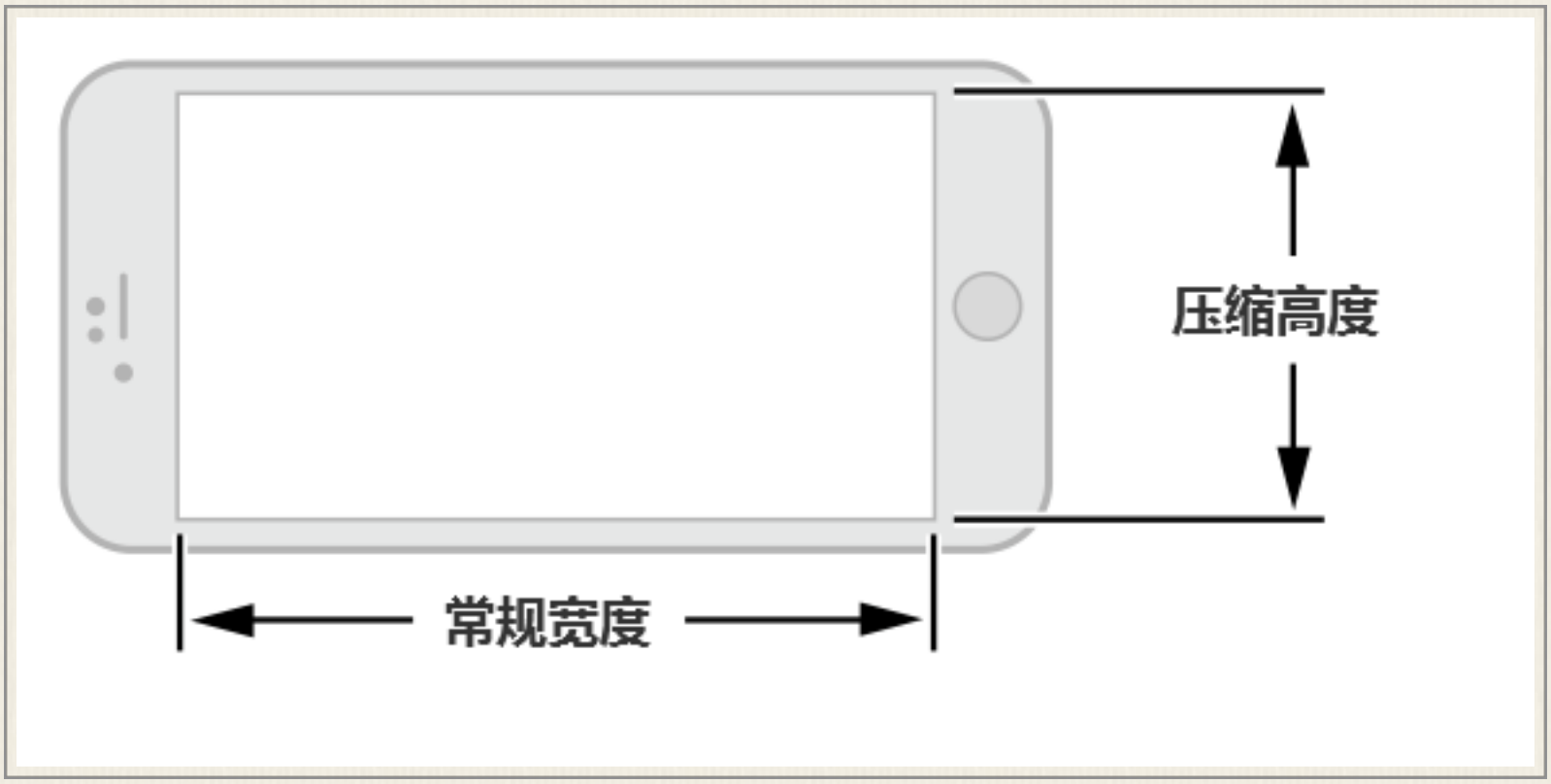


iPhone的显示环境可根据不同的设备和不同的握持方向而改变。

竖屏时，iPhone6 Plus使用的是常规高度和压缩宽度类型。

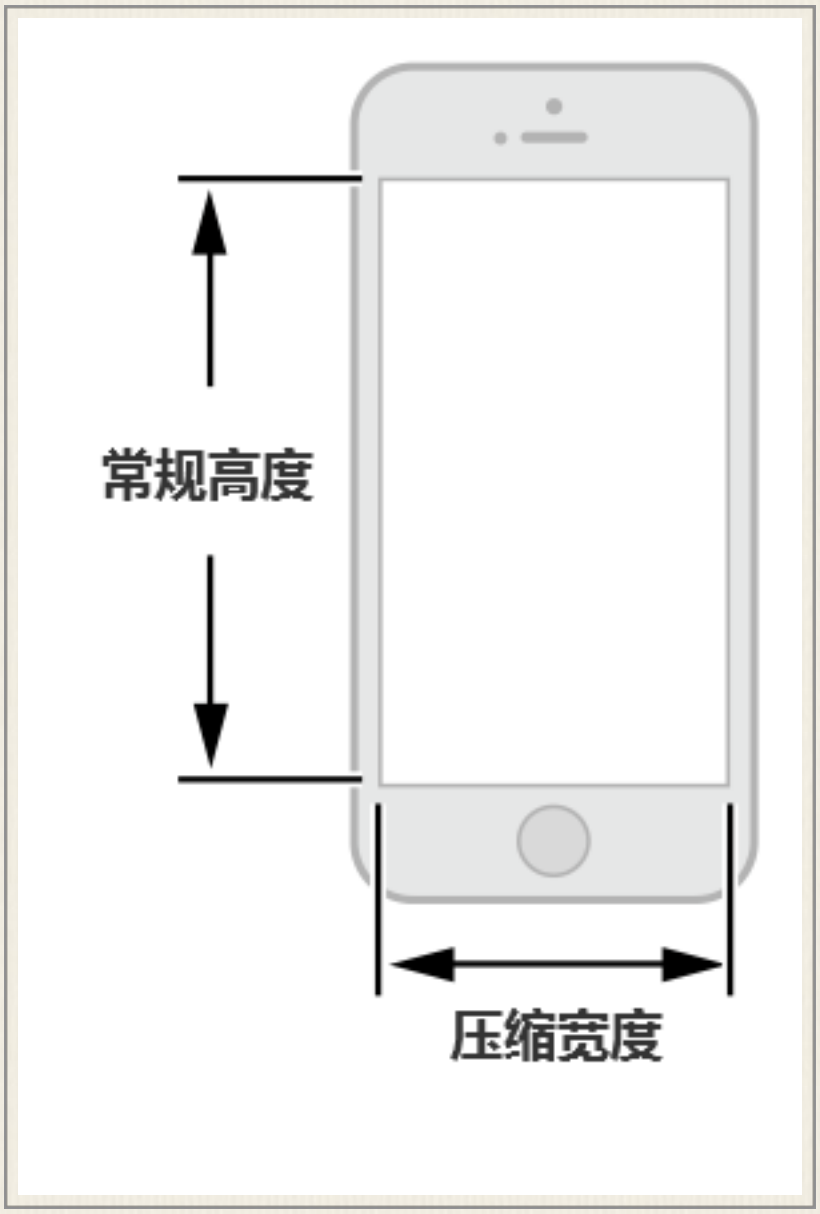


横屏时，iPhone6 Plus使用的是压缩高度和常规宽度类型。

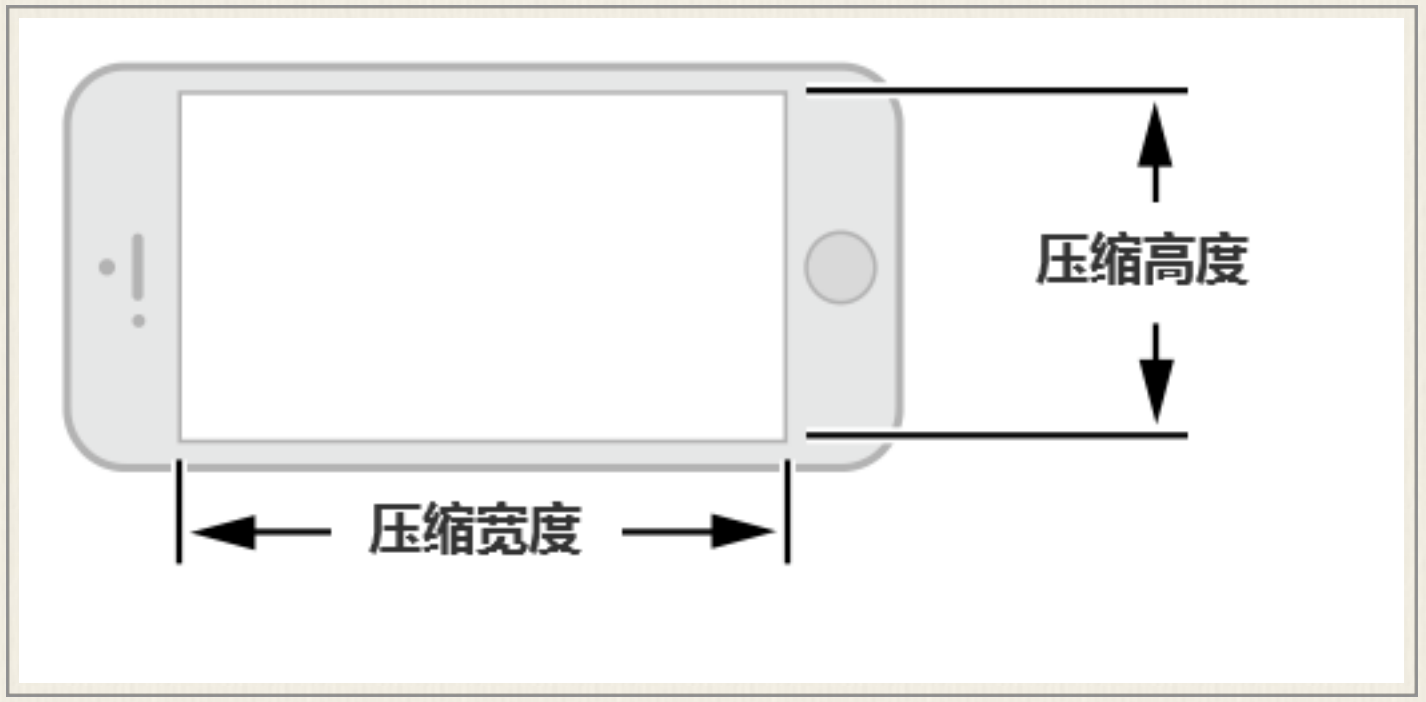


其他iPhone型号，包括iPhone6使用相同的尺寸类型设置。

竖屏时，iPhone 6，iPhone 5 和iPhone 4S使用的是压缩宽度和常规高度。



横屏时，这些设备在宽高上使用的都是压缩类。



1.3.2 在不同环境提供良好体验（Provide a Great Experience in Each Environment）

当你使用自适应来开发UI时，你可以保证UI跟随显示环境变化而适当地响应。遵照这些指南，你可以给用户良好的设备响应体验。

在所有环境下都保持对主体内容的专注。这是最高优先级。人们使用应用时，与感兴趣的内容发生互动。当使用环境变化的同时，改变专注点会让用户感觉到迷失方向，丧失了对应用的控制。

避免布局上不必要的变化。在所有环境中类似的使用体验让人们在旋转设备或在不同设备上运行你的应用时维持使用模式。例如，如果你使用一个网格在水平的常规模式下显示图像，你无需在一个列表中展示与压缩模式下相同信息，虽然你可能调整了网格的尺寸。

如果你的应用只在一个方向上运行，那么请直接一点。人们希望在各种握持方式下都可以使用你的应用，能满足这个期待是最好的。但是，如果你的应用只在一个方向下运行，那么以下几点请务必注意：

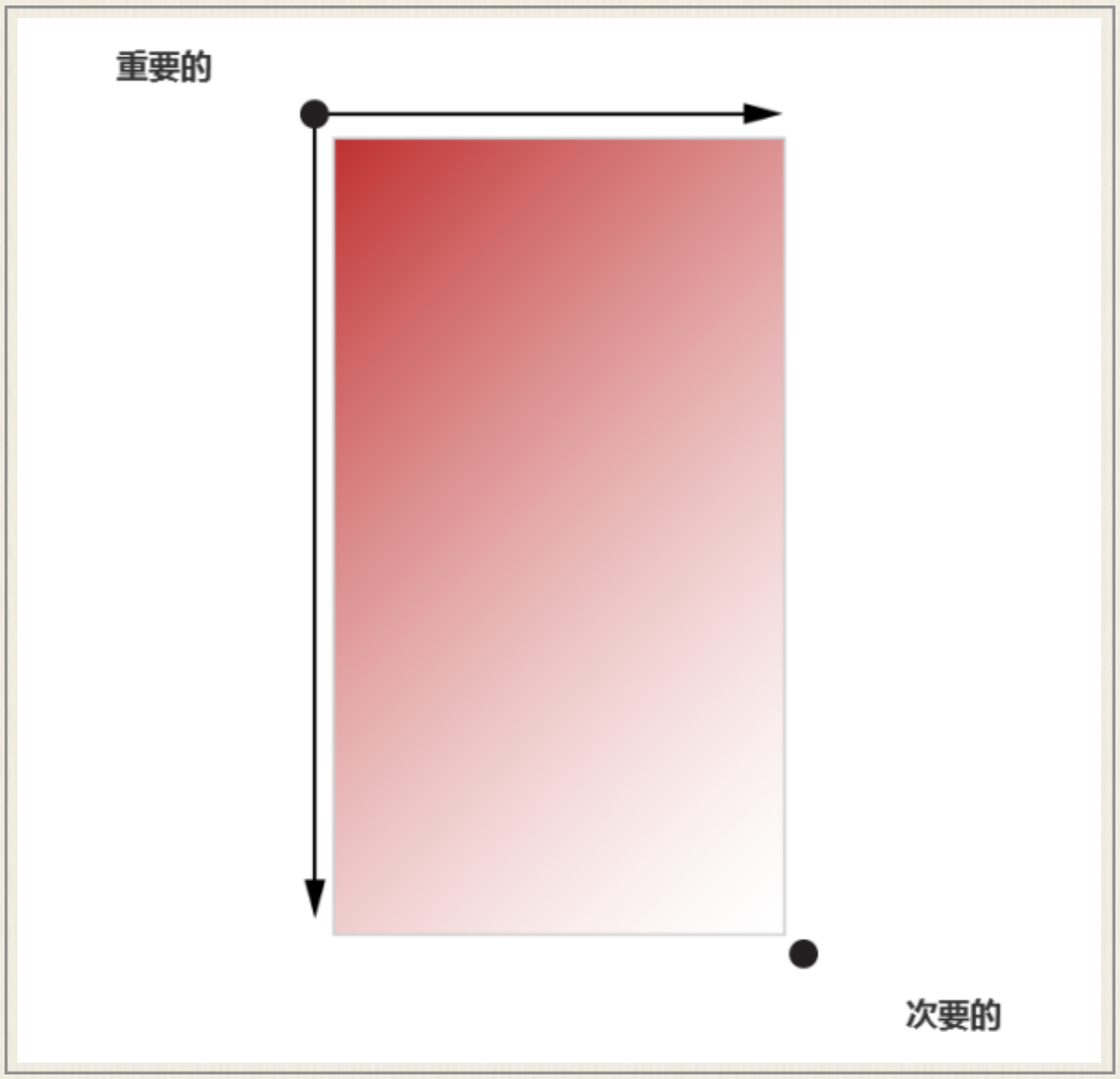
- 避免提示人们旋转设备的提示UI出现。让应用清晰地运行在支持的方向下，让用户明白应该旋转设备，而不是添加不必要的引导性混乱。
- 支持同一个方向上的变化。例如，如果一个应用只能垂直运行，用户无论用左手或是右手握持时都能触及到home键。如果用户在使用应用时180度旋转设备，那最好应用内容也能及时响应并旋转180度。

如果你的应用将设备方向翻转视为用户输入（的一种指令），那么就按照程序设定的方式来响应设备翻转。举 个例子，一个游戏让用户利用设备翻转来移动游戏中的部件，那么这个游戏应用本身（的UI）不能对翻转屏幕产生响应。在这种情况下，你必须关联两个需要变化 的方向，并且允许人们在这两个方向切换直到他们开始（了解并执行）应用的主体任务。一旦人们开始执行主要任务，那就开始按程序设定的那样来响应设备的动作 吧。

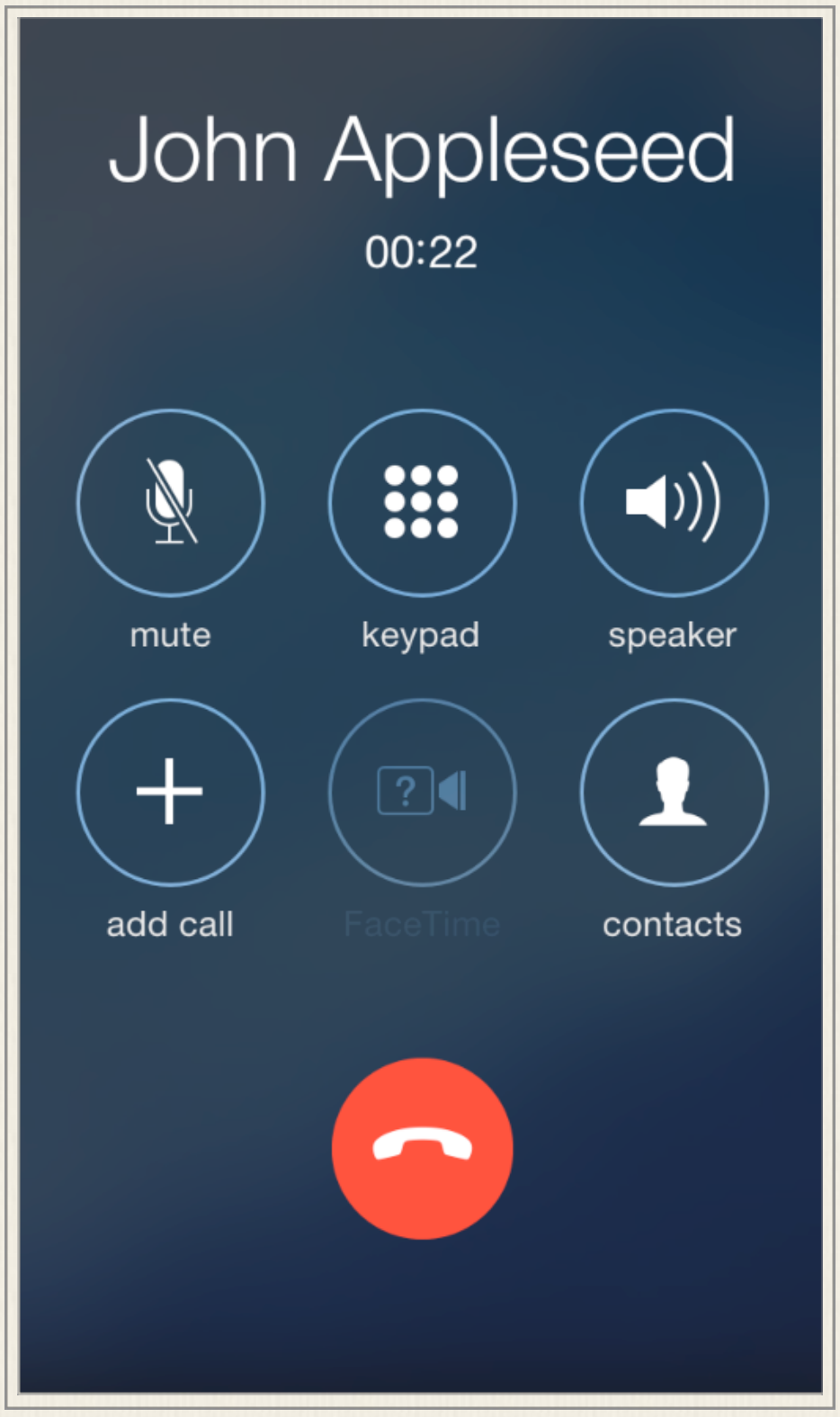
1.3.3 使用布局来沟通（Use Layout to Communicate）

布局包含的不仅仅是一个应用屏幕上的UI元素外观。你的布局，应该告诉用户什么是最重要的，他们的选择是什么，以及事物是如何关联起来的。

提升重要内容或功能，让用户容易集中注意在主要任务上。有几个比较好的办法是在屏幕上半部分放置主要内容，以从左到右的习惯，从靠近左侧的屏幕开始。


重要的
次要的

使用视觉化的重量和平衡向用户展示相关的屏显重要元素。大型控件吸引眼球，而比小的控件更容易在出现时被注意到。而且大型控件也更容易被用户点击，这让它们在应用中更加有用——就像电话和时钟（上面的按钮）——用户经常在容易分心的环境中能（正常）使用它们。



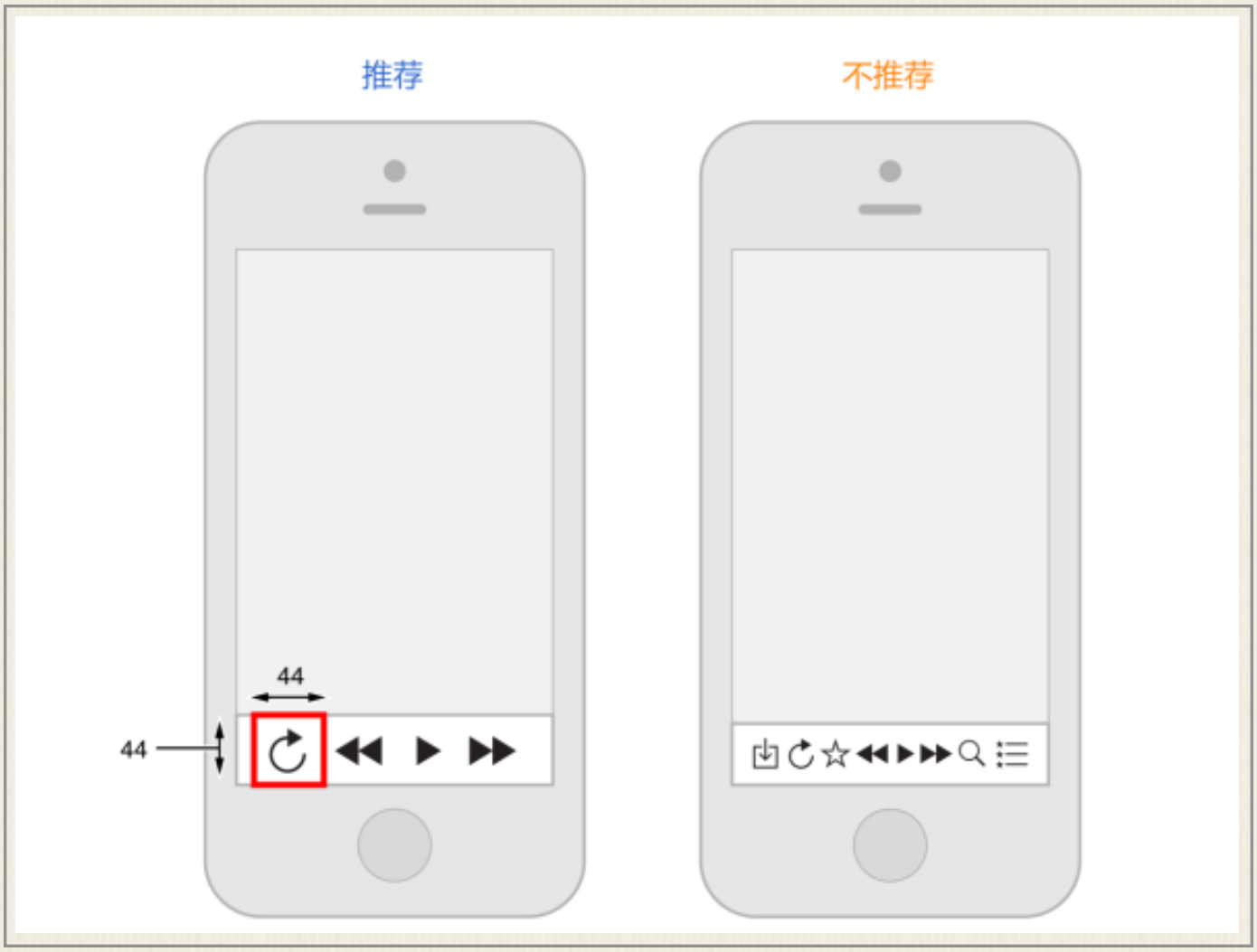
使用对齐来让阅读更舒缓，让分组和层次之间更有秩序。对齐让应用看起来整洁而有序，也让用户在专注于屏幕时更有空间，从而专注于重要信息。不同信息组的缩进与对齐让它们之间的关联更清晰，也让用户更容易找到某个控件。

确保用户能明白处于默认尺寸的首要内容的含义。例如，用户无需水平滚动就能看到重要的文本，或不用放大就可以看到主体图像。

准备好改变字体大小。用户期望大多数应用能有设置字号大小的功能。为了适应一些文本大小的变化，你也许需要调整布局；想要得到更多文本显示相关的信息，你可以查阅章节Text Should Always Be Legible.

尽量避免UI上不一致的表现。在一般情况下，有着相似功能的控件看起来也应该类似。用户常常认为他们看到的不同总有原因，而且他们倾向于花时间去尝试（译者按：因此为了避免用户做无用的尝试所以建议类似功能外观一样）。

给每个互动的元素充足的空间，从而让用户容易操作这些内容和控件。常用的点按类控件的大小是44 x 44点（points）。



1.4 起始与停止（Starting and Stopping）

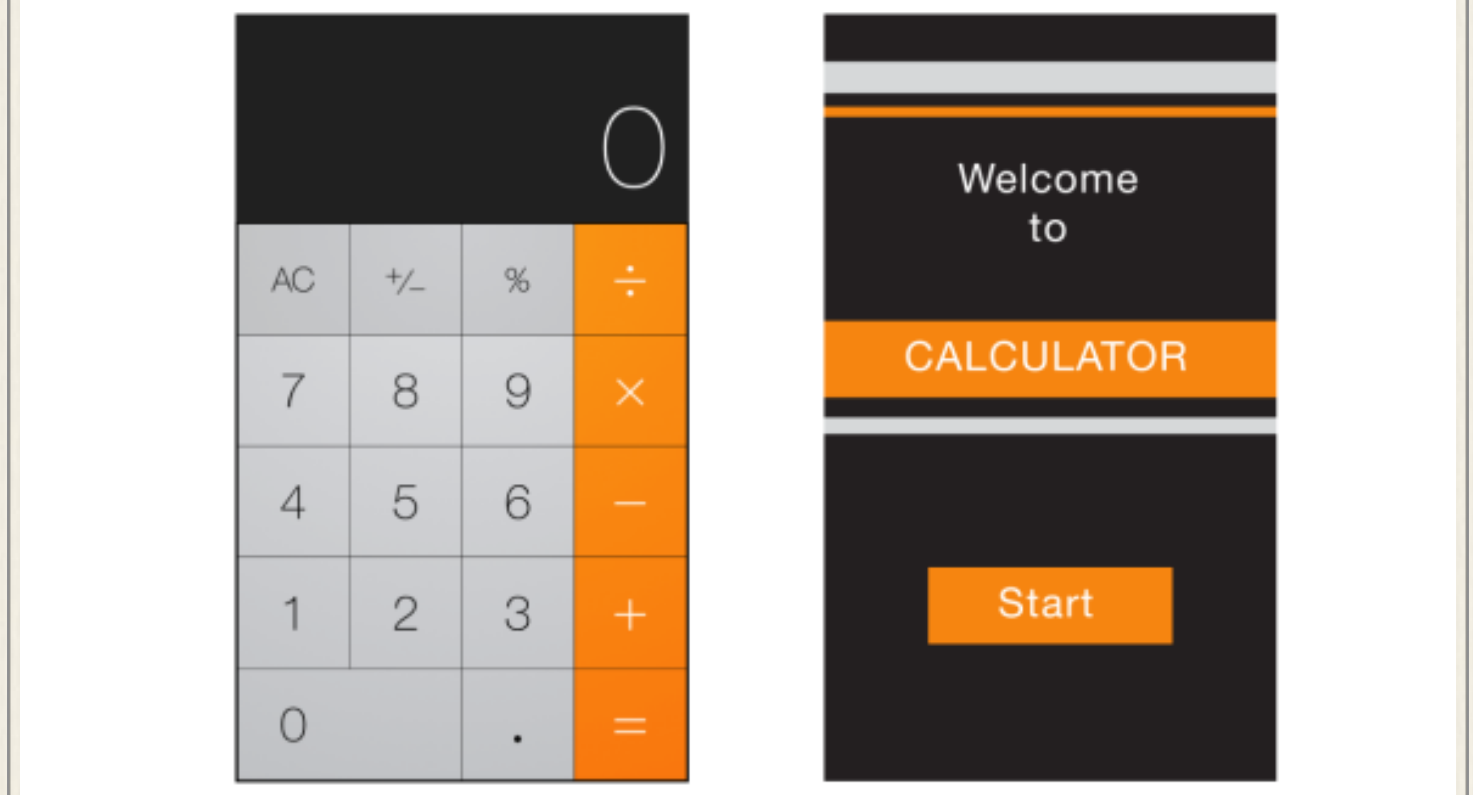
1.4.1 即时启动（Start Instantly）

有种说法是用户往往不会花超过一两分钟去审视一个新应用，当你将应用从打开到启动这段时间压缩得很短，并且同时在载入过程中呈现一些对用户有帮助的内容，你就会激发用户的兴趣并给所有用户一个惊喜。

重要：不要在安装过程结束后告诉用户需要重启设备。重启需要时间并且会让人觉得你的应用看上去不可靠而且很难使用。

如果你的应用有内存使用问题，或者不重启就无法流畅运行，你必须声明这些问题。关于如何开发一款性能良好的应用，请查阅iOS应用编程指南。

尽可能避免使用闪屏或者其他启动体验。用户能够在启动后立即开始使用应用是最好不过的。



避免让用户做过多设置。而应该如此：

- 聚焦在满足80%的用户需求上。这样主体用户群就无需设置各种选项，因为你的应用已经默认处于他们想要的状态。如果有些功能有少部分用户想要，换句话说，大部分人不需要的话，就别管它了。

- 尽可能用其他方式获取更多（用户）信息。如果你能得到用户在内置应用或硬件设置中提供的信息，直接从系统中获取它们，而不需要再次让用户输入。

- 如果你必须获取设置信息，在你的应用中直接向用户询问，然后尽快保存这些设定（译者注：这段讲的是权限许可，如能否访问照片或者日历或地理位置信息等等）。这样用户就无需强制跳出应用进入系统设置页面了。如果用户需要更改设置，他们可以在任何时候进入应用的设置选项进行修改。

尽可能让用户晚一些再登录。最理想的状态是，用户在无需登录的情况下就能尽量多地浏览内容并使用部分功能。例如，App Store应用会在用户

浏览商品并确定进行购买时，才要求用户进行登录。对于必须登录才能进行后续浏览和操作的应用，用户往往会直接放弃。

如果你的应用必须先登录后使用，那么你应该在登录页面有一些简短的文字，来描述为什么必须先登录，以及这样做会给用户带来什么好处。

谨慎使用新手引导（介绍应用的功能和如何进行操作）。在考虑新手引导之前，你应该完善你的应用，尽可能使应用的功能直观和易于寻找。有句话说得好，好的应用不需要新手引导。如果你确实觉得需要新手引导，那么请参考如下的建议，设计一个简洁、有针对性并且不妨碍用户的新手引导。

- 只提供开始使用应用所必需的信息。好的新手引导应该告诉用户接下来第一步应该做什么，或者简短地演示大部分用 户感兴趣的一些功能。在能够浏览你的应用之前，如果用户遇到太多的信息，让用户记住这些不是当前所必须的内容，他们很可能会觉得你的应用很难用。如果在某 些特定场景下确实需要一些引导，那么也应该在用户进入这个场景之后再进行。

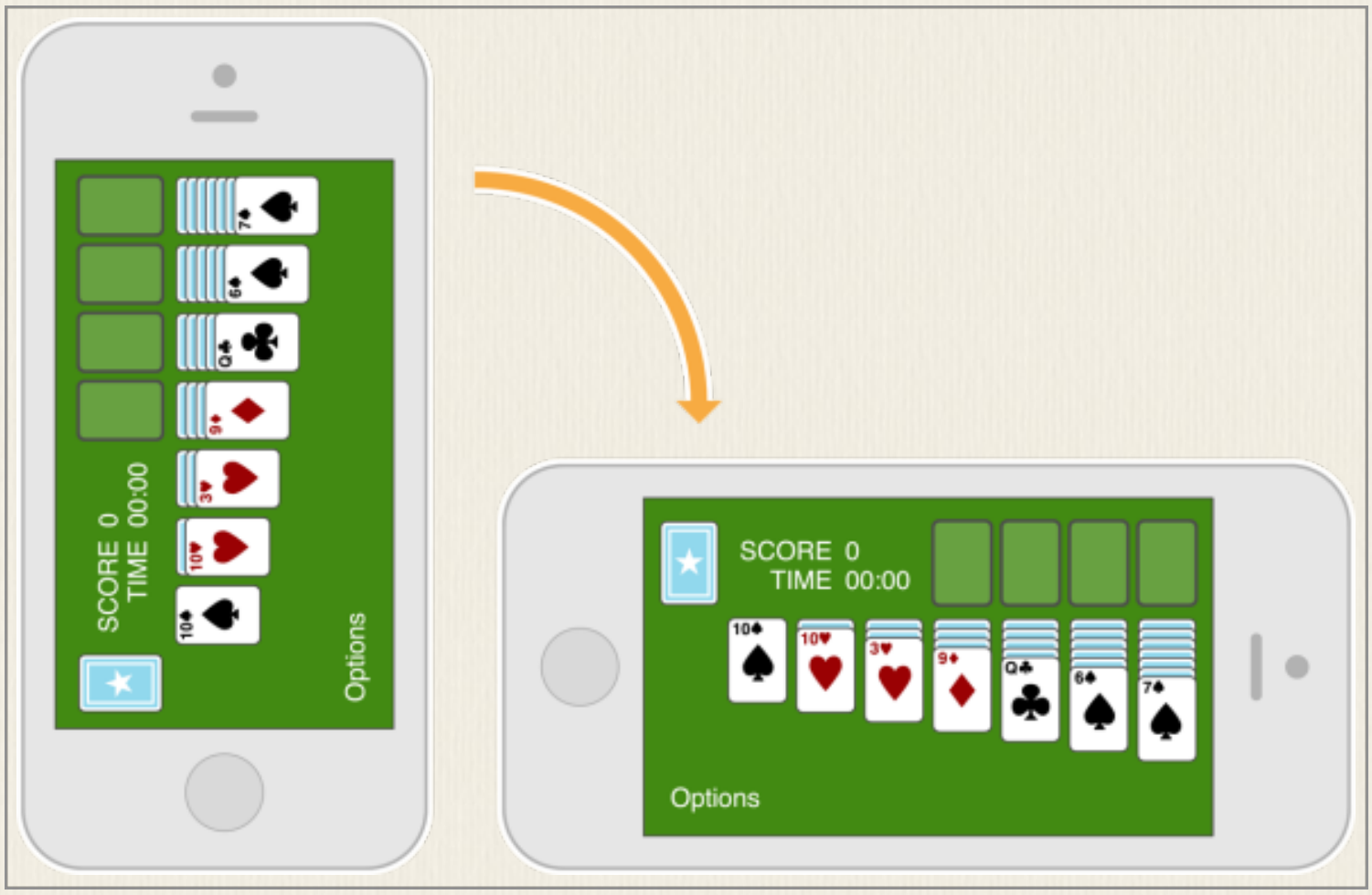
- 使用交互动画来吸引用户，并让用户通过实际上手来学习如何使用。对于文字内容的增加应该谨慎，且仅当增加文字 对于提升体验有益时才这么做。不要指望用户会阅读大段的文字。例如，可以使用动画而不是文字来描述如何执行一个简单的任务。在引导用户了解较为复杂的任务 时，可以通过一些引导浮层来简要说明每一个步骤用户需要做什么。尽可能避免展示应用截图，因为截图是死的，用户可能会混淆截图和应用的实际界面。

- 能够简单地取消或者跳过新手引导。有些用户看完新手引导之后就不想再看，有些甚至根本就不想看新手引导。请记住用户的选择，不要强迫用户每次打开你的应用都要再做一次选择。

不要太早要求用户去给你的应用评分。过早要求用户进行评分可能会适得其反。如果你想获得用户有价值的反馈和评论，在邀请用户评论之前，请给他们一点时间来使用你的应用，并对你的应用形成印象。例如，你可以等用户访问了一定数量的页面或完成了一定数量的任务之后，再邀请他们进行评价。

一般建议按照屏幕默认的定向方式启动你的应用。尽管如此，如果你的应用只有一种屏幕方向，那么就始终以这个方向 启动，让用户在他们自己需要时再改变设备方向。例如，一个游戏或者媒体观看应用只在横屏模式下

运行，那么就应该以横屏模式启动，即使设备当前处于竖屏模 式。这样的话，如果用户在竖屏模式下打开应用，他们也知道应该把设备转成横屏来进行使用。



注：最好让横屏应用支持两种模式的横屏，即home键 处于左右两侧的状态。如果设备当前已经处于横向状态，那么就按照当前状态启动应用，除非你有充分的理由不这么做。其他情况时，可以考虑按home键处于右 侧的方式启动应用。（想要了解更多关于支持不同设备方向的内容，请参阅Respond to Changes in Device Orientation.）

准备一张与应用首页看上去一样的闪屏。iOS会在启动应用时调用这张图，这样可以让用户觉得启动速度很快，降低对等待时间的感知度。具体如何制作闪屏，请查阅Launch Images。（译者注：Launch Images章节处在iOS Human Interface Guidelines的Icon and Image Design部分，翻译将在后续更新中放出，烦请各位耐心等候。）

如果可能，不要让用户在初次启动应用时阅读免责声明或者确认用户协议。你可以直接在App Store展示这些内容，使用户在下载前就有所了解；虽然这个办法能最大地减少麻烦，但也不是一直可行。如果在某些情况下你必须展示这些内容，要确保它们与UI保持统一并在产品功能与用户体验之间达成平衡。

在应用重启后，需要恢复到用户退出使用时的状态，让他们可以从中断之处继续使用。无需让用户记住是如何达到此种退出状态的。

1.4.2 时刻准备好停止（Always Be Prepared to Stop）

iOS 应用无需关闭或退出选项。当用户切换应用，回到主屏幕或者将设备调至睡眠模式的时候，其实就是停止了当前应用的使用。

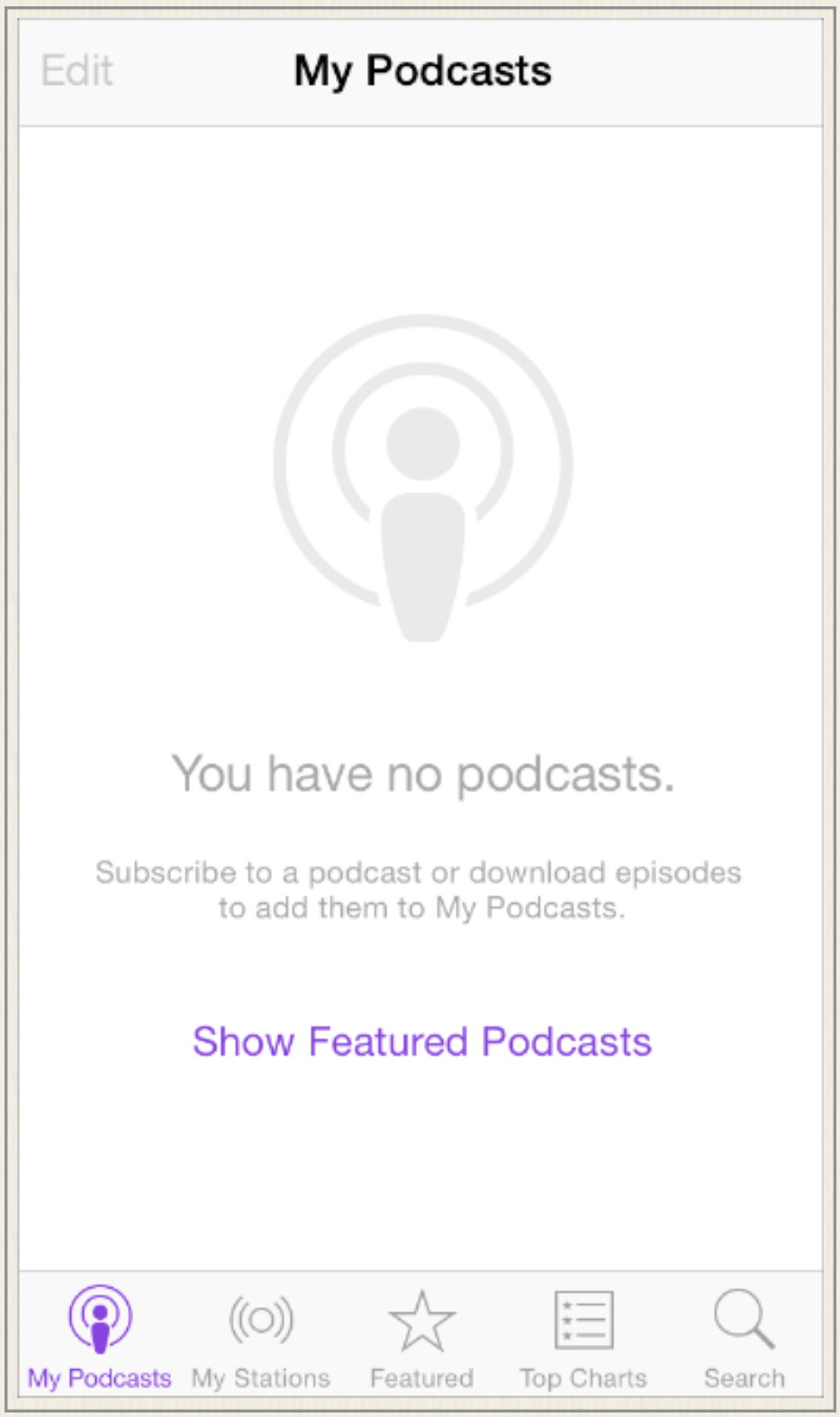
当用户切换应用时，iOS的多任务系统会将其放置到后台并将新应用的UI替换上来。在这种情况下，你必须做到以下几点：

- 随时并尽快保存用户信息，因为在后台的应用随时有可能被终止或退出。
- 当应用停止的时候保存当前状态，使用户可以在回到应用时能从中断之处继续使用。例如，在使用可滚动的数据列表时，退出后保存列表所在的位置。

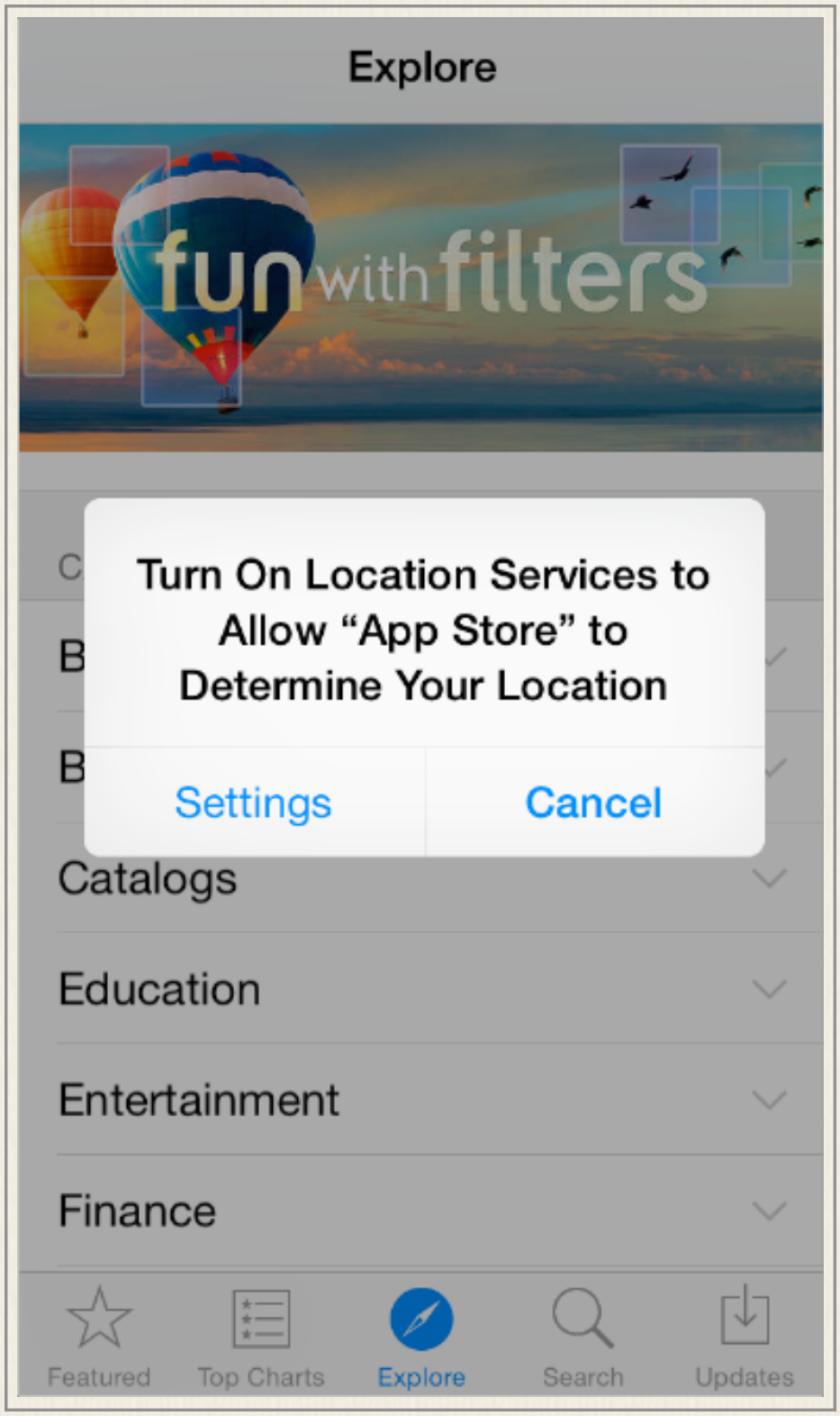
有些应用可能需要一直在后台运行。例如，用户可能希望能在使用一个应用的同时还能一直听歌，接着又想用另外一个应用来检查代办项或者玩游戏。关于如何正确处理多任务，请查阅Multitasking。（译者注：Multitasking章节处在iOS Human Interface Guidelines的iOS Technologies部分，翻译将在后续更新中放出，烦请各位耐心等候。）

不要强制让应用退出。因为这样会让用户误以为是crash。如果有问题产生，需要告诉用户具体状况以及如何解决。以下有两个建议，取决于出现的问题有多严重而酌情使用：

如果应用中所有的功能当前都不可用，那么应该显示一些内容来解释当前的情形，并建议用户如何进行后续操作。这部分内容给予了用户以反馈，使用户相信你的应用现在没问题。同时这也可以稳定用户情绪，让他们决定是否要采取纠正措施，继续使用应用，还是切换到另一个应用。



如果只有部分功能不可用，那么只要当用户使用这些功能时显示提示即可。不然的话，用户就应该能正常使用应用的其他功能。如果你决定使用警告框来进行提示，请确保只在用户尝试使用不可用的功能时再显示。



1.5 导航（Navigation）

除非导航设计的不合理，不然用户不应明显察觉到应用中的导航体验。放置导航到一个能够支撑你的应用结构和目的却又不过分引起用户注意的状态。

广义来说，有三种主要类型的导航，每种导航都有其适应的应用结构：

- 分层。
- 扁平。

- 内容或经验驱动。

在分层应用中，用户在每个层级中都要选择其中一项，直到目的层级。如果要切换到另一个层级，用户必须回退一些层级，或者直接回到初始层级进行再次选择。系统的设置和邮件应用在这方面是很好的示范，可以参考他们。

在扁平应用中，用户可以从一个主要分类直接切换到另一个，因为所有的主要分类都可以从主屏直接访问。音乐和App Store是两个使用扁平结构的好例子。

在内容驱动或经验驱动信息结构的应用中，导航的内容也会根据内容或经验来进行设计。例如，在阅读一本电子书时，用户会一页接一页地进行阅读，也会在目录中选择想要阅读的页码跳转后开始阅读。同样的，在游戏应用中，导航的作用也非常重要。

在某些情况下，在一个应用中结合多种导航类型会有很好的效果。例如，对于扁平信息结构中某一分类下的内容，用分层导航的方式来显示可能会更好。

用户应该时刻清楚自己当前在应用中所处的位置，以及如何前往他们所想到的页面。

无论导航类型是否适合你的应用结构，最重要的是用户访问内容的路径应该是合理、可预期和易于寻找的。

UIKit定义了一些标准的UI元素，这些元素即可以构建分层或扁平的导航，也可以实现以内容为中心的导航，例如电子书或者媒体观看类应用。游戏或者其他经验驱动的应用通常需要一些自定义的元素和行为。

使用导航栏（Navigation Bar）帮助用户轻松访问分层内容。导航栏的标题可以显示用户当前所处的层级，而后退按钮可以回到上一层级。查看Navigation Bar了解更多。

使用标签栏（Tab Bar）显示同类型的内容或功能。标签栏很适合于扁平信息结构，可以让用户在分类之间随意切换，而不用在意当前所处的位置。查看Tab Bar了解更多。

在应用中，如果每屏显示的都是同类项或同类页，可以使用页面控件（Page Control）。页面控件的优点是可以直观地告诉用户共有多少个项目或页面，以及当前所处的位置。查看Page Control了解更多。

一般来说，最好能给用户到达每一屏的路径。如果用户需要，就应该考虑使用临时视图，例如模态视图、动作菜单或警告框。查看Modal View、Action Sheet和Alert了解更多。

（译者注：上文提到的章节均处在iOS Human Interface Guidelines的第4章，翻译将在后续更新中放出，烦请各位耐心等候。若有需要，亦可先参考先前已翻译的iOS7 UI Elements章节：上，下。）

UIKit同时还提供了以下相关控件：

- 分段控件（Segmented Control）。分段控件让用户在一屏内就可以查看到不同分类的内容，而不需要切换到其他屏幕。

- 工具栏（Toolbar）。尽管工具栏看起来和导航栏或标签栏相似，但是工具栏不具导航作用。相反，工具栏为用户提供了可以控制当前屏幕内容的控件。

1.6 模态情境（Modal Contexts）

模态是一个承载某些连贯操作或内容的优缺点并存的模式。它可以给用户提供一种不脱离主任务的方式去完成一个任务或者获得信息，但是也会临时性地阻止用户对应用的其他部分进行交互操作。

理想情况下，用户可以与iOS 应用进行一种非线性的交互，所以，尽可能减少应用中的模态体验是最好的。通常情况，在以下情形下可以考虑使用模态情境：

- 必须引起用户关注的时候。
- 一个独立的任务需要完成或者很明确需要被放弃，为了避免在模棱两可的状态下遗漏用户信息的时候。

保持模态任务的简单，简短和高度聚焦。你肯定不希望用户使用模态视图时像使用应用中的一个mini应用一样。如 果子任务过于复杂，用户会在进入模态情境时忽略主要任务。在设计一个涉及视觉层次的模态任务时特别要考虑这一点，因为用户有可能迷失并且忘记如何回到之前 的操作中去。如果一个模态任务必须包含不同视图的子任务，确保给用户一个独立、清晰的导航路径，并避免迂回。

始终提供明显、安全的途径退出模态任务。确保用户在退出模态视图时可以预期操作的结果。

一个任务需要多层级的模态视图时，确保用户理解点击完成按钮的结果。点击一个低层级视图上的完成按钮是完成这个视图中任务的一部分，还是整个任务？因为存在这种困惑的可能性，所以要尽可能避免在下级视图中添加完成按钮。

保证提醒对话框的内容都是重要且可操作的。提醒对话框会打断用户的体验并且要点击才会消失，所以要让用户感到提醒信息是有用的，打断是有价值的。

尊重用户关于接收通知的选择。用户会设置接收应用通知的形式，必须尊重要用户的喜好设置，否则可能触怒用户，导致其关闭所有的推送通知。

1.7 交互性和反馈（Interactivity and Feedback）

1.7.1 用户知道标准手势（Users Know the Standard Gestures）

用户使用点击、拖拽、捏合等手势与iOS设备进行交互。使用手势拉近了用户和设备之间的距离，并且增强了直接操纵感。用户通常期待手势在不同应用之间都是通用的。

除了用户熟悉的标准手势，iOS还定义了一些系统范围内的操作，例如呼出控制中心或消息中心。在任意应用下都可以使用这些操作。

不要给标准手势赋予不同的行为。除非你的应用是游戏，否则重新定义标准手势会使用户迷惑，且增加使用难度。

不要创建和标准手势功能相似的手势操作。用户已经习惯了标准手势的行为，没有必要让用户学习达到同样效果的不同操作。

可以用复杂手势作为完成某任务的快捷方式，但不能是唯一触达方式。最好给用户提供一些简单、直接的方式完成某操作，即使这种方法需要额外的动作。简单的手势能让用户集中于当前的体验和内容，而不是交互操作本身。

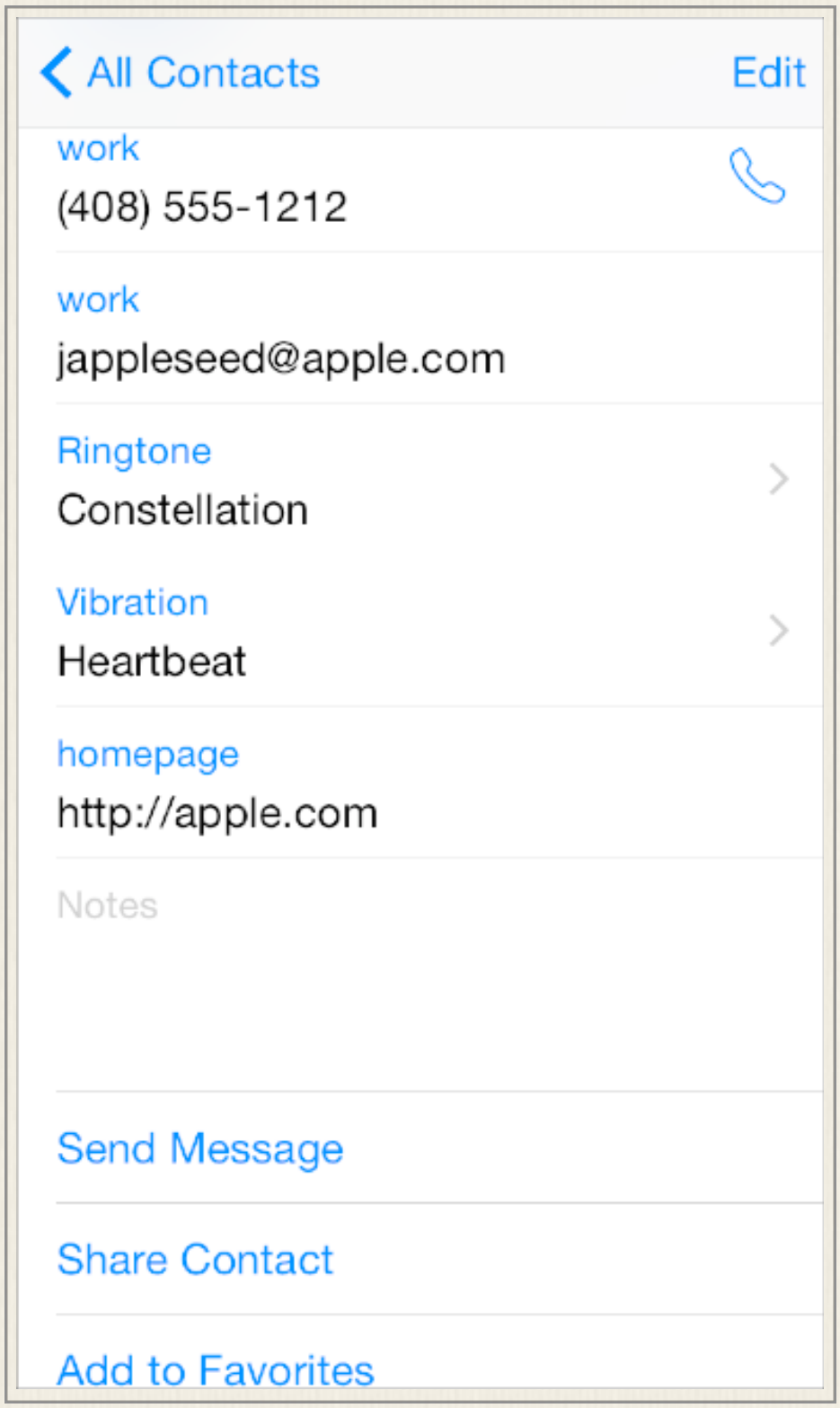
除非是游戏，否则避免定义新的手势。在游戏或其他沉浸式的应用中，操作手势也是有趣体验的一部分。但是在普通应用中，帮助用户达成目标要比操作本身重要的多，所以最好使用标准手势，尽量避免让用户去发掘和记忆新的操作。

在特定的环境中，可以考虑使用多指操作。虽然复杂的操作并不一定适用于所有应用，但对用户会花大量时间使用的应用来说可以丰富体验，例如游戏或创建内容环境。但因为非标准手势可发现性差，要尽量少用，并且不要让这类手势成为完成任务的唯一方式。

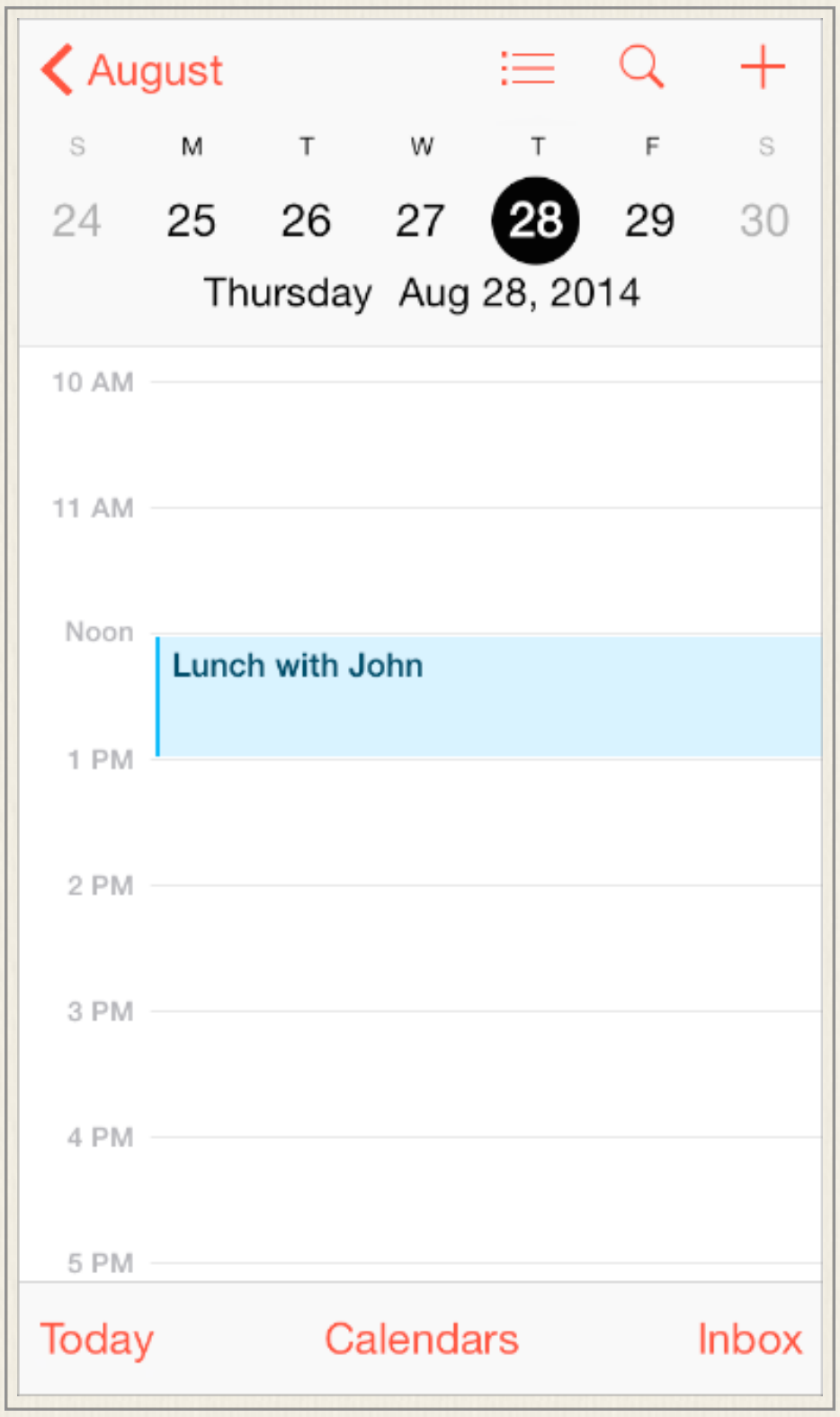
1.7.2 可交互元素吸引用户点击（Interactive Elements Invite Touch）

为了暗示交互性，设计时会使用很多线索，包括颜色、位置、上下文、表意明确的图标和标签等。并不需要过于修饰元素来向用户展示可交互性。

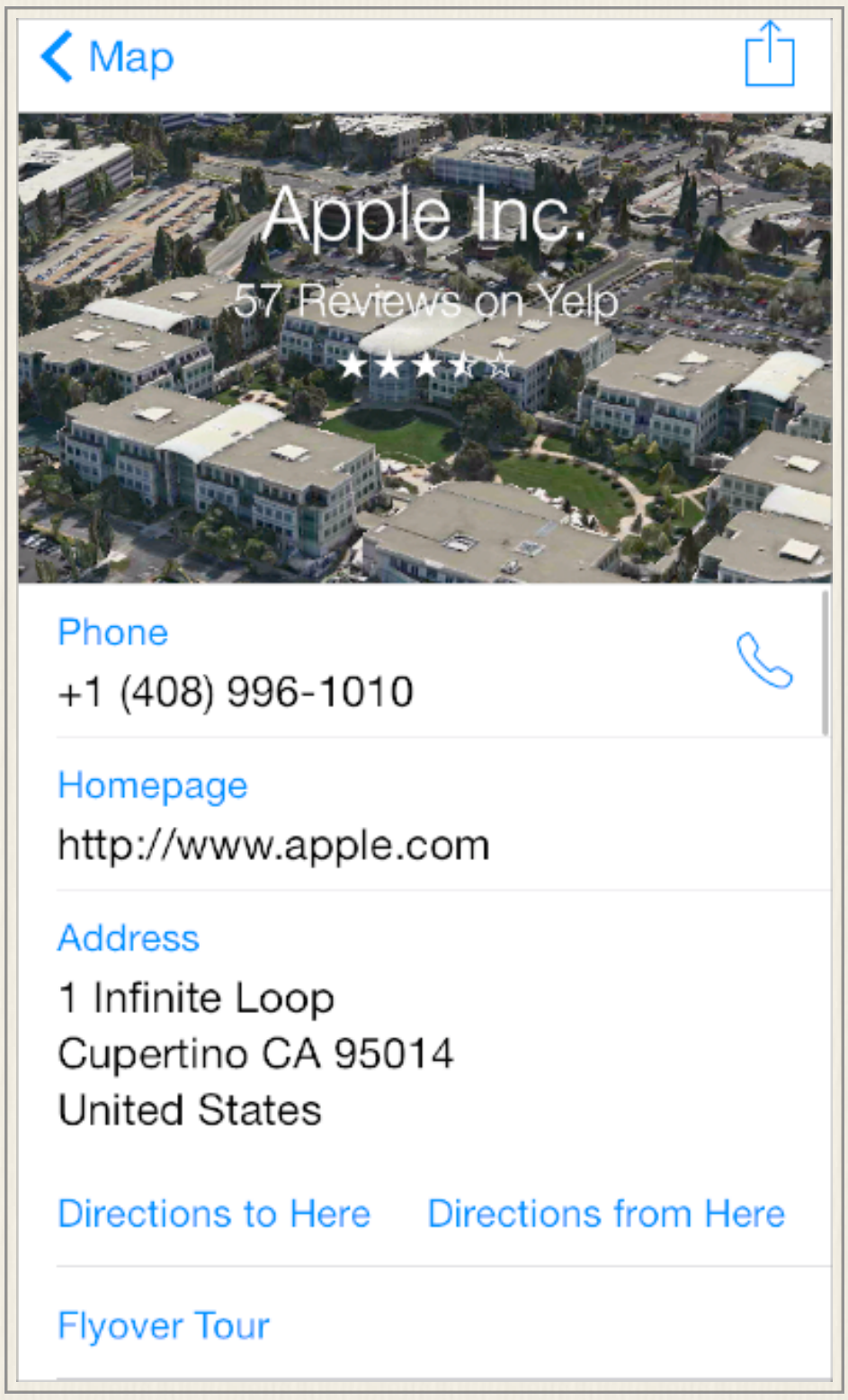
一个关键的颜色可以给用户提供很强的视觉指引，尤其是没有冗余的其它颜色时。为了有对比，使用蓝色标记可交互的元素，并且使用统一的、易识别的视觉风格。


All Contacts
Edit
work
(408) 555-1212
work
jappleseed@apple.com
Ringtone
Constellation
Vibration
Heartbeat
homepage
http://apple.com
Notes
Send Message
Share Contact
Add to Favorites

返回按钮使用多个线索指明其可交互性并传达其功能：它出现在导航中，显示了一个指向后方的图标，使用了关键色，显示了上一级页面的标题。



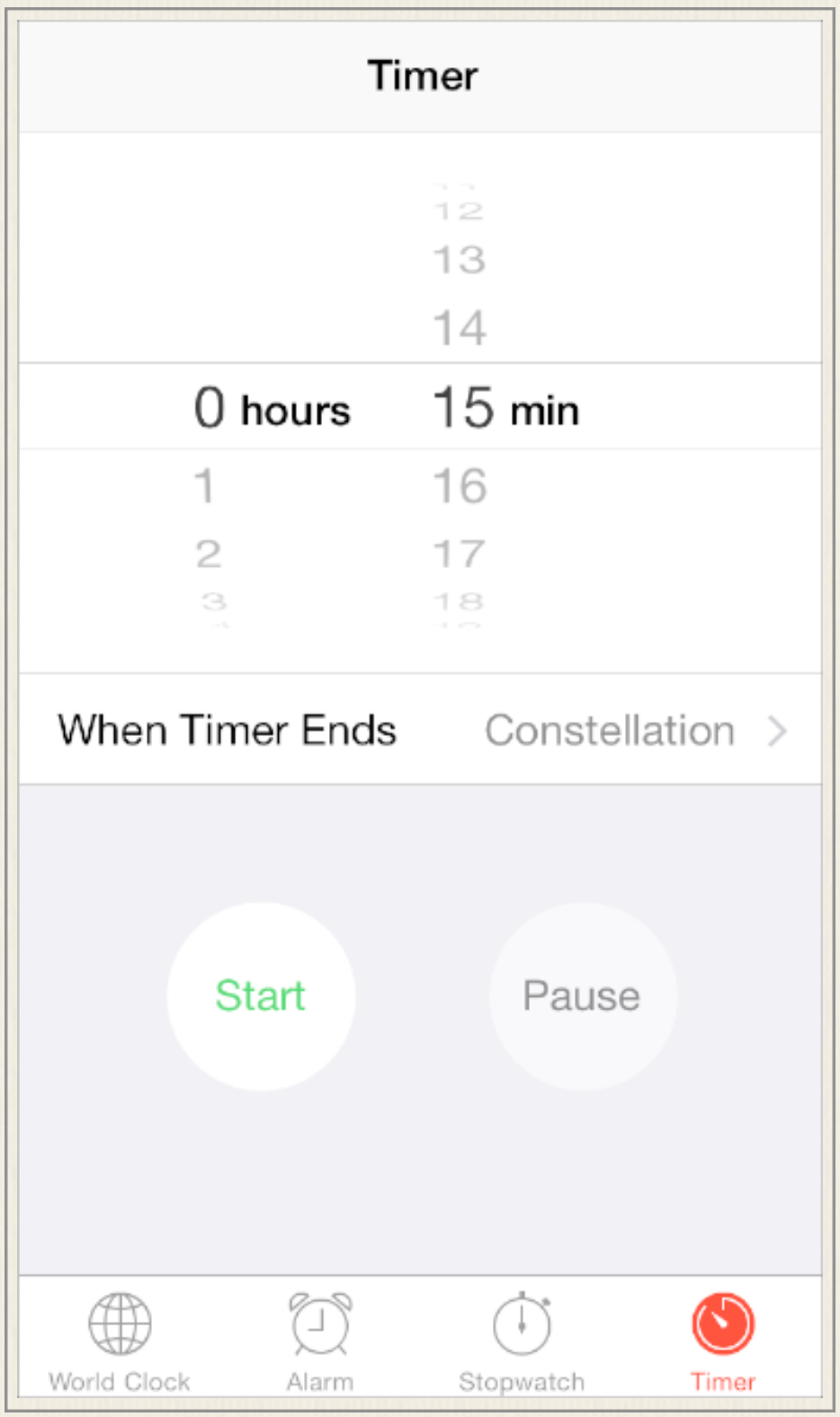
一个图标或者标题提供了清晰的名称指引用户点击它。例如，地图中的标题“立交桥路线”，“定位到这里”，清晰地说明了用户可做的操作。结合关键色，就可以省去按钮边界或其他多余的修饰。



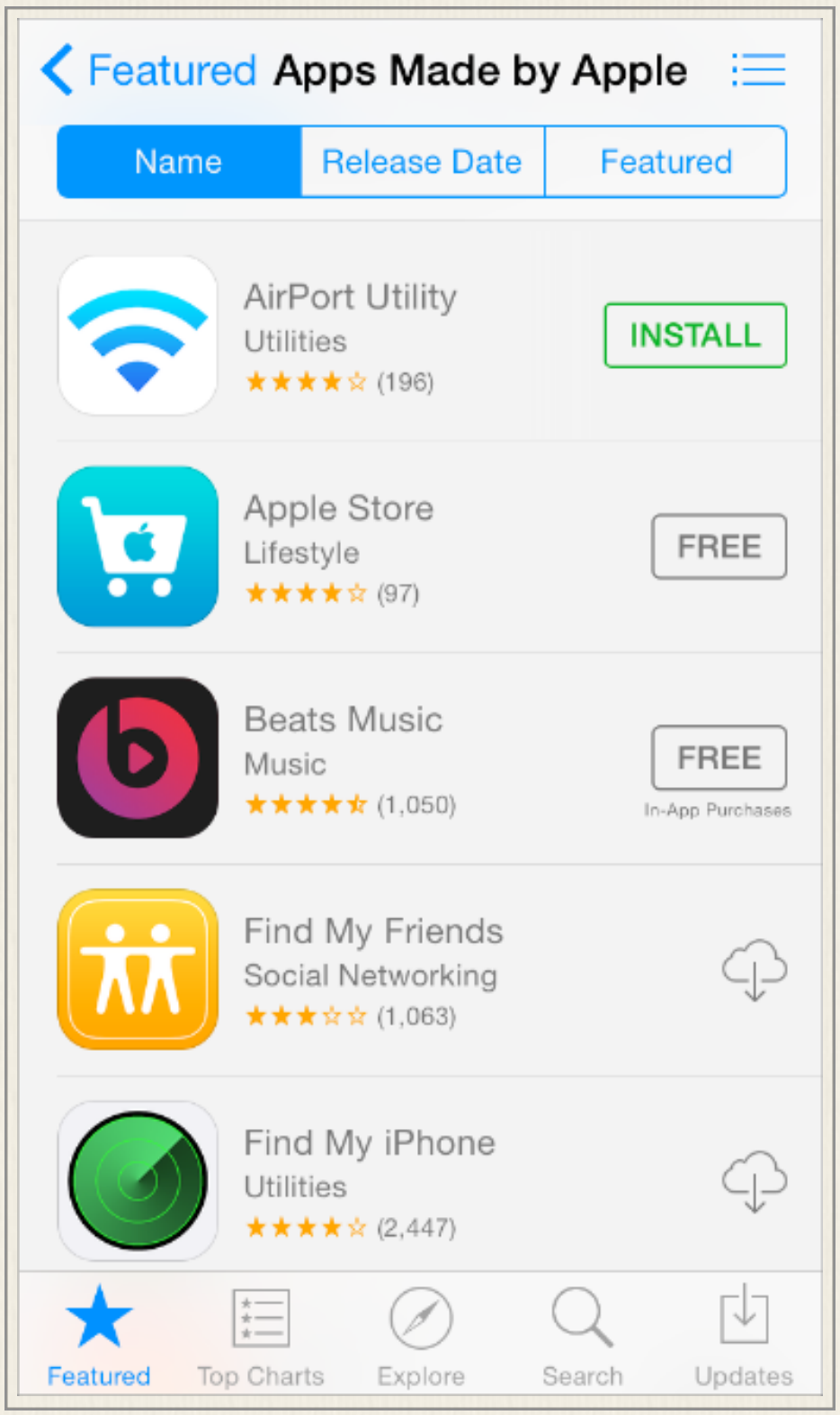
在内容区域，有必要给按钮添加边界或背景。操作条中的按钮、动作表单和提醒对话框可以不需要边界，因为用户知道在这种区域中大多数选项是可交互的。但是在内容区域，按钮有必要使用边界或背景将按钮从其他内容中区分出来。例如，系统自带的音乐、时钟、照片和App Store应用会在一些特别的场景中使用这种按钮。

照片应用中给分享按钮增加了边框，与其他解释性文本进行了区分。

时钟应用在秒表和计时页面中给按钮增加背景来强调开始和暂停按钮，并且使按钮在周围的内容中更容易点击。



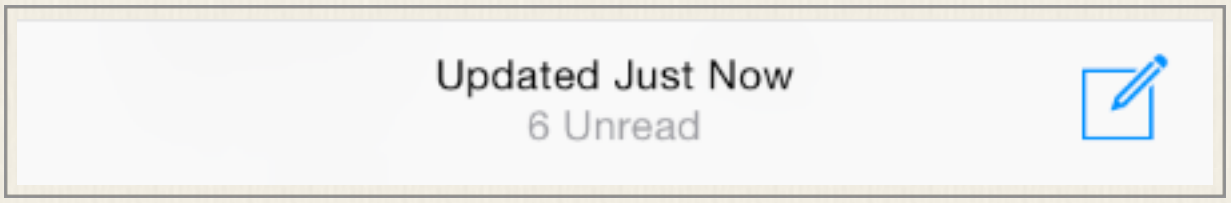
App Store应用中使用有边界的按钮，将按钮和整个内容条区分开来，点击整条内容查看详细信息，点击按钮进行下载安装。



1.7.3 反馈有助于理解（Feedback Aids Understanding）

反馈会帮助用户了解应用当前在做什么，发现接下来可以做什么以及理解动作产生的结果。UIKit提供了很多反馈。

尽可能将状态或其他的反馈信息整合到UI中。用户不进行操作或不跳出当前内容就能获得需要的信息是最好的。例如，邮箱应用将当前邮箱的状态显示工具条上，这样就不会影响当前内容。



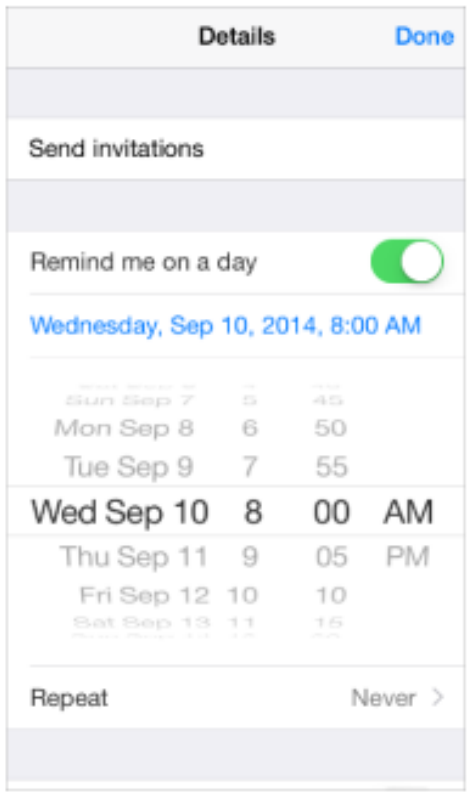
避免显示不必要的警告框。警告框是一种很强的反馈机制，只有在传递非常重要也是理论上可行的信息时才需要使用它。如果用户常看到很多不是重要信息的警告框，他们很快就会忽略所有对话框提醒。

1.7.4 输入信息的方式要简单（Inputting Information Should Be Easy）

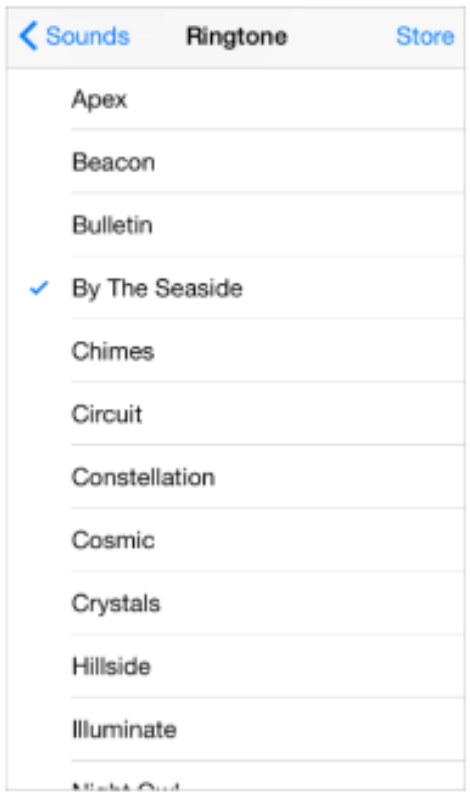
不管用户是点击控件还是使用键盘，输入信息都会花费时间和精力。如果发挥有用的效用前就让用户输入大量信息会减弱用户继续使用的欲望。

让用户更容易地进行选择。例如，使用选择器或者表格代替纯文本，避免要求用户打字来提高选择效率，降低选择成本。

提醒中的时间日期选择器



设置中的一列选项



适宜地从iOS中获取信息。设备上存储了大量的用户信息。可以的话，不要让用户提供你可以轻易找到的信息，例如联系人或日历信息。

提供有用的反馈来平衡用户的输入。付出和回报的概念可以帮助用户感到进程在被推进。

1.8 动画（Animation）

iOS的用户界面中遍布着细微、精美的动画，它们使得应用的体验更具吸引力、更具动态性。适当的动画可以：

- 传达状态和提供反馈
- 增强直接操作的感觉
- 帮助人们可视化他们的操作结果

谨慎地增加动画，特别是在那些无法提供沉浸性体验的应用中。看起来过多的无理由的动画会阻碍应用的流畅性，降低 性能，还会分散用户在任务中的注意力。尤其要说的是，要有目的和限制性地使用运动效果和UI组件中的动态行为，并确保对结果进行测试。一旦被合理的使用， 这些效果能

提高用户的理解度和愉悦度；过度使用他们则会使应用看起来很迷惑，很难控制。

在合适的时候，使自定义的动画与内置动画保持统一。人们习惯于谨慎添加动画，尤其是在那些不能提供沉浸式用户体验的应用中。如果应用主要关注一些严肃的任务或者生产性任务，那么动画就显得多余了，还会无端打乱应用的使用流程，降低应用的性能，让用户从当前的任务中分心。

开发者的自定义动画应该切合内置iOS应用的动画。用户习惯于内置iOS应用使用的精细动画。事实上，用户趋向于把视图之间的平滑转换，对设备方向改变的流畅响应和基于物理力学的滚动效果看作是iOS体验的一部分。除非你的应用能够给用户沉浸式的体验–比如游戏–自定义动画应该可以与内置应用的动画相媲美。

使用风格类型一致的动画。在应用中使用风格类型一致的动画非常重要，可以让用户构建基于使用应用获得的用户体验。

大多数情况下，恰当一点的做法是让自定义动画更具现实性。用户乐意于接受自由的艺术创作，但当你的动画违背物理定律和自然法则的时候，他们会感觉到非常迷惑。

1.9 品牌推广（Branding）

品牌推广并不仅仅是在应用中展示品牌的颜色和logo。理想状态下，你开发的某个特定品牌的应用应该通过创建独特的外观和感觉来为用户提供难忘的体验。

在iOS系统之下可以很容易地使用自定义的图标、颜色和字体来创建区别于其他应用的UI。当你进行这些元素的设计时，牢记以下两点：

- 每个自定义的元素本身都需要具备良好的观感和功能性，但它也应该与应用中其他元素保持一致，无论应用中其他元素是自定义的还是标准的。

- 为了在iOS中感觉舒适，你的应用虽然不必看起来跟内置的一样，但是需要对它的遵从、清晰度和深度（如欲了解更多，参见1.1 为iOS而设计（Design for iOS））进行整合。花些时间弄清楚在你的应用中遵从、清晰和深度所代表的意味，并把它们在你的自定义元素中表达出来。

当你需要让用户意识到你的品牌时，你应该遵循以下几点：

以精致优雅不唐突的方式植入品牌的颜色和图片。用户使用你的应用来完成事务或者进行娱乐，他们不希望被强迫着去观看广告。为了获得最好的用户体验，你可以通过字体、颜色和图像的设计来潜移默化地地提醒用户你的品牌身份。

避免远离用户关心的内容。比如，在屏幕顶部展示一个二级栏目，仅用来展示品牌资产，这意味着内容没有足够的空间，可以考虑以其他低侵入性的方法无处不在地展示品牌，比如巧妙地定制屏幕的背景。

抵抗住诱惑，不要把你的logo贯穿整个应用。移动设备的屏幕多数相当小，logo的每一次出现都会占据空间而 将用户与他们想看的内容隔离开。而且，在应用中显示logo并不能像在网页中显示logo那样达到相同的目的：对于用户来说通常会很容易在不知道网页所属 的情况下访问一个网页，但却极少有用户会在完全不看一个iOS系统中的应用图标的情况下就打开它。

1.10 颜色与字体（Color and Typography）

1.10.1 色彩有助于增进沟通（Color Enhances Communication）

在iOS系统中，颜色会用于表征交互，传递活性以及提供视觉连续性。内置的应用程序选择使用那些看起来更具个性的、纯粹、干净的颜色，并辅以或亮或暗的背景组合。

如果你要创建多样的自定义颜色，要确保它们能够和谐共存。例如，如果你的应用的基本风格是柔和的色调，你就应该创建一个协调的柔和色调的色板用于整个应用。

注意在不同情境下的颜色对比。例如，如果在导航栏的背景与栏按钮标题之间没有足够的对比，按钮就会很难被用户看到。 依据经验的法则来说，需要区分的颜色必须至少存在50%的亮度差异。（我们）需要将设备置于不同的光照环境之中（包括晴朗的室外）来测试设备上的观感效果。

提示：一种发现需要更高对比度的区域的方法是降低UI的饱和度并在灰度模式下查看它。如果在灰度版本中你很难区分可交互与非可交互元素或背景等，你有可能需要增加这些元素之间的对比度。

当你使用自定义的栏颜色时，着重考虑半透明的栏和应用内容。当你需要创建能匹配特别颜色的栏颜色时（比如一个已有品牌中的颜色），可能在

你获得你想要的结果之前，你需要用各种颜色进行实验。栏的显示将会同时受到iOS系统所提供的半透明栏与藏在栏后面的应用内容的呈现所影响。

API注释：使用浅色（TintColor）的属性值给予栏按钮颜色，使用栏浅色（BarTintColor）的属性值为栏本身赋色。欲了解更多关于栏属性的内容，可参见UINavigationBar Class Reference,，UITabBar Class Reference，UIToolbar Class Reference和 UISearchBar Class Reference。（译者注：相关章节翻译将在后续更新中放出，烦请各位耐心等候。）

注意颜色的盲区。多数色盲的人很难区分红色与绿色。需要对你的应用进行测试以确保在其中你没有将红色与绿色作为 区分两个不同状态或值的唯一方式，一些图像编辑软件或工具能够有效的帮你验证颜色的盲区。通常意义来说，使用多种方式来表征原色的交互性是非常好的（需要 了解更多关于在iOS系统中表征交互性的信息，详见Interactive Elements Invite Touch）。

考虑选择一种基准色颜色来表征交互性与状态。在内置的应用中基准色有比如备忘录中的黄色与日历中的红色等。如果你定义一种用于表征交互和状态的基准色，要确保你的应用中的其他颜色不会与它发生冲突。

色彩可以向用户传达信息，但不一定会以你希望的方式。每个人眼中的色彩是不一样的，不同的文化为色彩赋予的意义也是不相同的。花时间来研究如何使用色彩才可能会被其他国家或者文化接受。你要尽可能确定应用中运用的色彩向用户传达了恰当的信息。

大多数情况下，不能让颜色喧宾夺主，让用户分心。除非色彩是应用的目的和本质所在，通常情况下色彩应该用来从细微细节之处提升用户体验。

1.10.2 文字应该清晰易读（Text Should Always Be Legible）

文字首先必须是清晰可辨的。如果用户不能看清楚应用中的字词，那么文字再好看也是没是无意义的。当你在你的应用中采用Dynamic Type时，你可以实现：

- 能自动调整文字的粗细，字母间距以及行高。
- 为语义上有区别的文本模块指定不同的文本样式，比如正文、脚注或者标题。

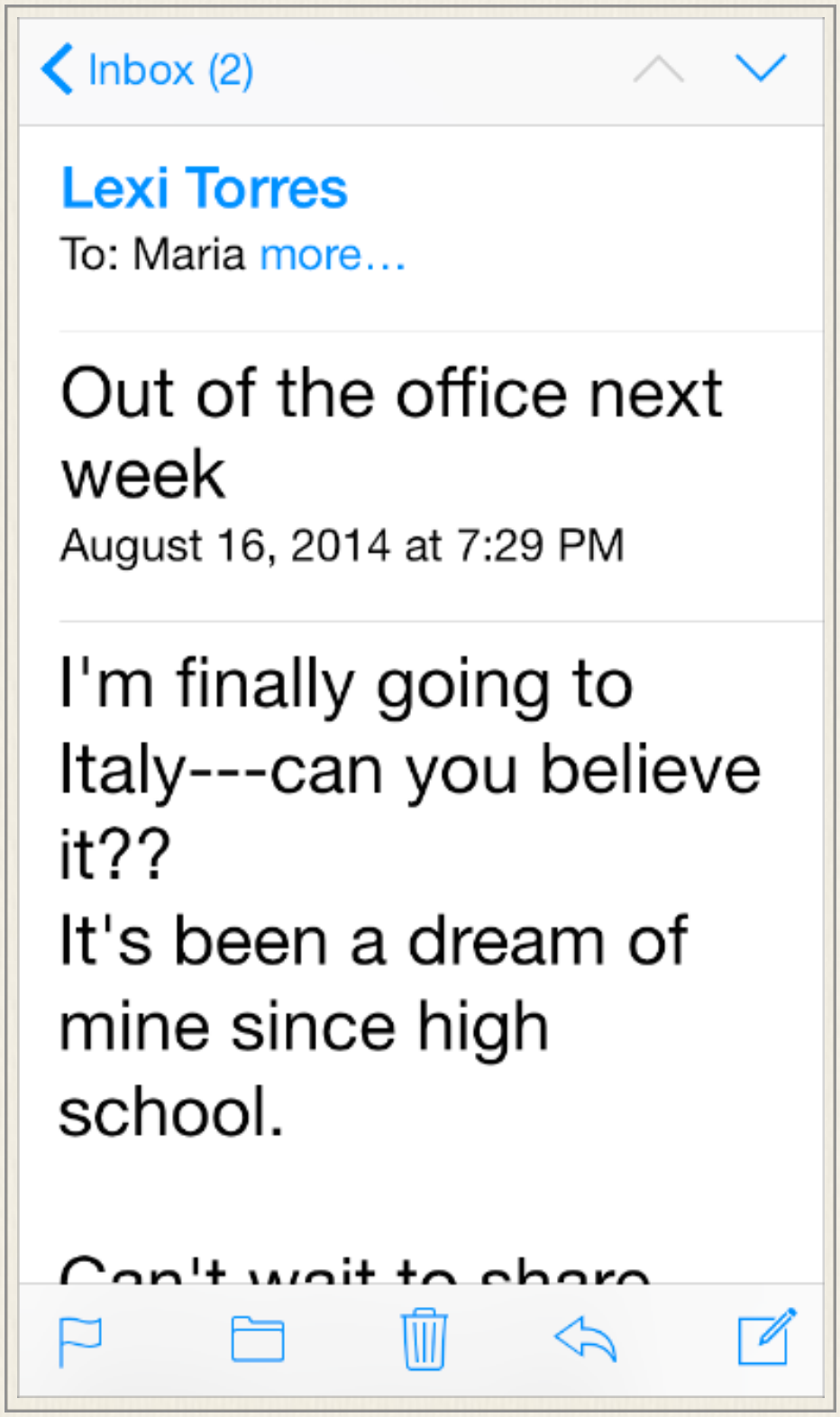
- 文本可以根据用户在动态文字和可访问性设置中指定字体大小的变化作出适当的响应。

注：如果你是用自定义字体，你仍然可以依据系统的字号设置来规划字体范围。当用户改变设置时，你的应用也必须响应式的配合。

就你而言，要采用Dynamic Type需要一些工作。为了学习如何使用文字样式并确保当用户改变文字型号设置时你的应用能够获取通知，可以参考Text Styles in Text Programming Guide for iOS。（译者注：相关章节翻译将在后续更新中放出，烦请各位耐心等候。）

文本尺寸的响应式变化需要优先考虑内容。并不是所有的内容对于用户都是同等重要的。当用户选择更大的文本尺寸时，他们是想要使他们关注的内容更容易阅读；他们并不总是想要屏幕上的每个单词都更大。

例如，当用户选择具备更大易用性的文本尺寸时，邮件将会以更大的尺寸显示邮件的主题和内容，而对于那些没那么重要的信息——如时间和收件人——则采用较小的尺寸。



在适当的情况下，当用户选择一个不同的文本尺寸时要调整页面布局。例如，当用户选择小的文本尺寸时，你可能想将内容由一列的布局方式改为两列。如果你决定根据不同的文本尺寸调整布局，你可以选择针对尺寸的子集来实现——如包含小，中和大尺寸——而不是对于每个可能的尺寸都进行布局的调整。

确保一个自定义字体在不同尺寸下的所有类型都具备可读性。实现这一效果的方法之一是效仿在不同的文本尺寸下iOS系统呈现字体样式的一些方法。例如：

- 文本永远都不应该小于11点（points），即使是用户选择极小的文本尺寸。相较而言，内容样式使用17点的字号作为大尺寸的默认文本尺寸设置。

- 通常来说，字号与行距值在每一档的文本尺寸设置中差别为1点。唯一例外的是两种标题的样式，它们被应用在极小、小和中尺寸的设置中，使用了相同的字号、行距和字距。

- 在最小的三种文本尺寸中，字间距相对较大；而在最大的三中文本尺寸中，字间距相对紧凑。

- 标题和内容的样式使用相同的字体尺寸，同时，为了区分标题与内容样式，标题样式使用更重的值。

- 导航控制栏的文本使用相同的字号，而内容文本的样式则使用大尺寸的设置（值为17点）。

- 文本总是使用常规或者中重，一般不适用轻或者加粗。

通常情况下，应用中整体应该使用单一字体。多种字体的混杂会使你的应用看上去支离破碎和草率。相反，使用一种字体和少数样式。根据语义用途，使用UIFont类的API来定义不同文本区域的样式，比如正文或者标题。

1.11 图标和图形（Icons and Graphics）

1.11.1 应用图标（The App Icon）

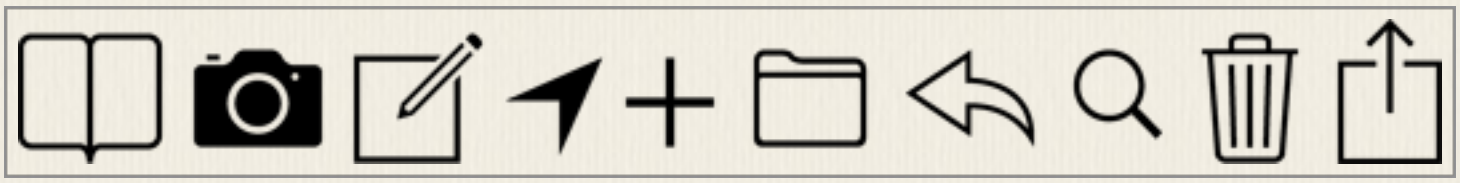
每个应用都需要一个漂亮的图标。用户常常会在看到应用图标的时候便建立起对应用的第一印象，并以此评判应用的品质、作用以及可靠性。

以下几点是你在设计应用图标时应当记住的。当你确定要开始设计时，请参考App Icon来获取更详细的设计规格与指导。（译者注：App Icon章节处在iOS Human Interface Guidelines的Icon and Image Design部分，翻译将在后续更新中放出，烦请各位耐心等候。）

- 应用图标是整个应用品牌的重要组成部分。将图标设计当成一个讲述应用背后的故事，以及与用户建立情感连接的机会。
- 最好的应用图标是独特的，整洁的，打动人心的，让人印象深刻的。
- 一个好的应用图标应该在不同的背景以及不同的规格下都同样美观。为了丰富大尺寸图标的质感而添加的细节有可能让图标在小尺寸时变得不清晰。

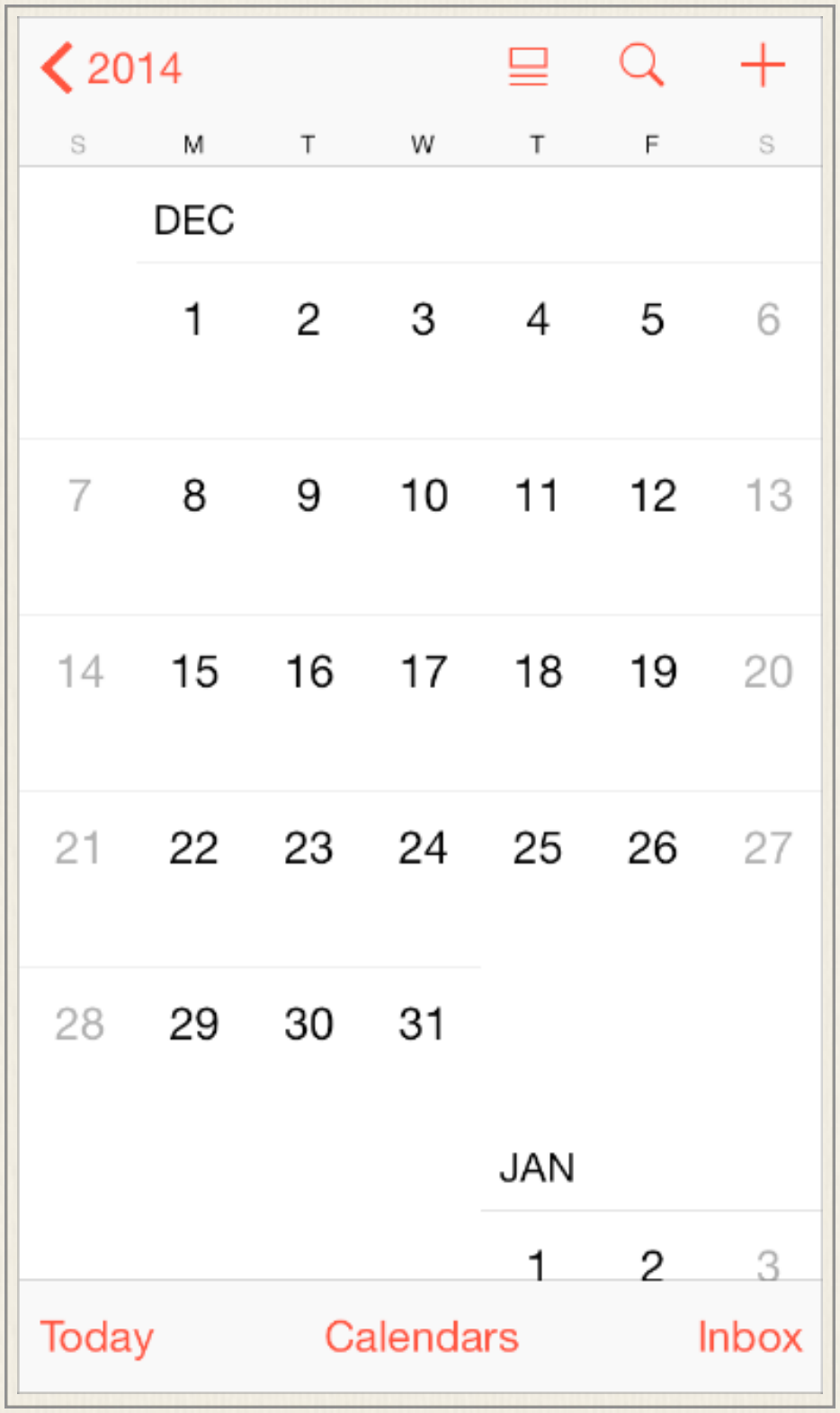
1.11.2 栏图标（Bar Icons）

iOS提供了一系列小的icon，用以代表各种常见任务与操作，它们常用在标签栏(Tab Bar)、工具栏(Toolbars)与导航栏(Navigation Bar)中。用户通常都已经了解这些内置图标的含义了，因此可以尽可能的多使用它们。



如果需要自定义动作或者内容，你也可以设计自定义图标。设计这些小的线性图标与设计应用图标有很大的区别，请参考Bar Button Icons来了解更多内容。（译者注：Bar Button Icons章节处在iOS Human Interface Guidelines的Icon and Image Design部分，翻译将在后续更新中放出，烦请各位耐心等候）

请注意，你有时候也可以用文字来代替工具栏和导航栏的图标。 就像iOS的日历里面，工具栏上便是使用“今天”、“日历”和“收件箱”来代替图标进行表意的。



想要决定在工具栏和导航栏中到底是用图标还是文字，可以优先考虑一屏中最多会同时出现多少个图标。如果数量过多，可能会让整个应用看起来难以理解。使用图标还是文字还取决于屏幕方向是横向还是纵向，因为水平视图下通常会拥有更多的空间，可以承载更多的文字。

1.11.3 图形（Graphics）

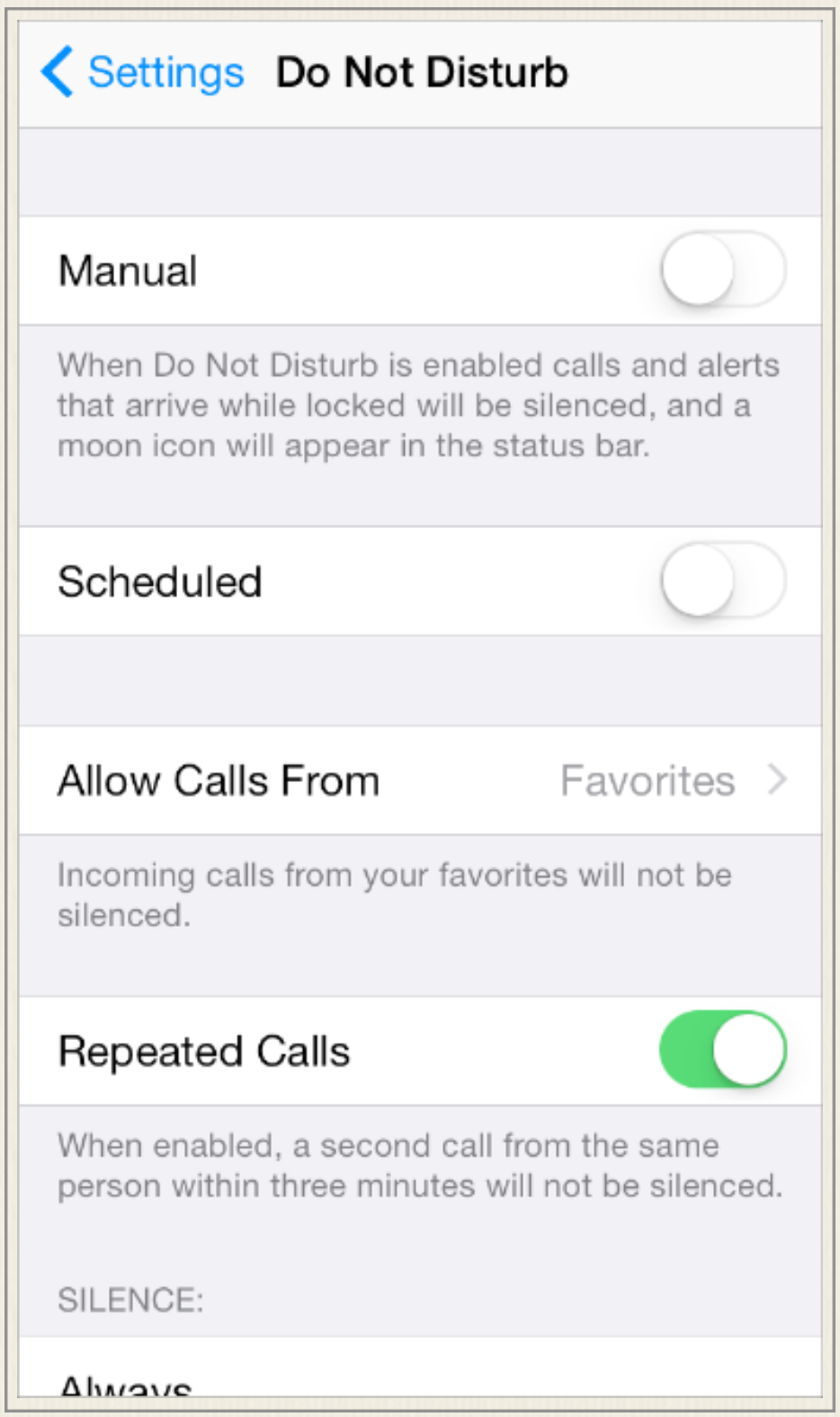
iOS应用大多数图形丰富。无论是你需要展示用户的照片，还是需要创建自定义图片，以下这些需求都应该遵守：

- 支持Retina显示屏。确保你应用中的所有图片资源都提供了高分辨率规格。尤其需要注意的是，iPhone 6 Plus需要提供@3x规格的图片，而所有其他的高分辨率iOS设备都需要提供@2x规格的图片。
- 显示照片或图片时请使用原始尺寸，并不要将它拉伸到大于100%。你不会希望在你的应用中看到拉伸和变形的图片。可以让用户自己来选择他们是否想要缩放图片。

不要使用带有苹果符号与版权的图片。这些符号都拥有版权，并且产品的设计可能会经常改变。

1.12 术语和措辞（Terminology and Wording）

你在应用中呈现的每一个字都是与用户进行对话的一部分。把握这样的对话机会，为你的用户提供清晰的表意与愉悦的体验。



设置是面向全体用户的一个基础应用，它使用了简明扼要的语言来描述了用户可以进行的操作。举个例子，设置→勿扰模式（Do Not Disturb）就没有使用难以理解的复杂术语，而是用了简单的语言，给用户描述了里头的一系列操作。

保证你使用的术语是用户能理解的。根据你对用户群的理解来决定在应用中使用什么样的词汇。举个例子，在一款针对小白用户的应用中使用技术术语是不合适的，但对于针对高端用户的应用来说，使用技术术语是很自然的事情。

使用非正式的友好语气，但不需要太过低三下四。避免太正式太僵化，或者太过嘻嘻哈哈，傲慢无礼。请记住，用户可能会反复阅读这些文本，因此有些起初看上去很俏皮的语句，多看几次之后可能会显得幼稚和烦人。

像新闻编辑一般遣词造句，避免不必要的冗余语句。当你的文案足够简明扼要，用户就可以很轻松地阅读和理解它。确定最重要的信息，精炼它并且突出它，让用户不需要读一大段文字才能了解他们在找什么，以及下一步要做什么。

给控件加上短标签或者容易理解的图标。让用户只扫一眼就能知道这个控件是干什么的。

描述时间时要注意准确性。今天和明天这些词汇确实显得比较友好，但有时候会让用户费解，因为你可能没有办法确定 用户当前的时区和时间。举个例子，假如有一项活动会在半夜12点前开始，对于在同一个时区的用户而言，这个活动是在今天开始的，但对于那些在早一点的时区 里的用户而言，这个活动在昨天就已经开始了。

为你的应用写一则漂亮的App Store描述，最大程度地把握住这个与潜在用户沟通的绝佳机会。除了准确描述你的应用、强调应用的品质与亮点以外，你还需要：

- 修正所有的拼写、语法与标点符号错误。这些小错误也许不会影响用户正常使用，但是可能会让他们对应用的整体品质产生负面印象。
- 尽量少用全大写的词汇。虽然大写单词有时候可以吸引注意力，但是全大写的段落不适合阅读，而且会产生一种朝用户扯着嗓子吼的感觉。

- 可以描述bug修复情况。如果你的应用新版包含用户一直期待的bug修复，那在你的软件描述中提到这一点就是个很好的做法。

1.13 与iOS的整合（Integrating with iOS）

与iOS整合，指的是在当前平台上给用户提供一种舒适的、宾至如归般的体验，当然这并不意味着我们要把每一个应用做的和内置应用一模一样。

最好的与iOS整合的方式便是深刻地了解iOS的主题与核心——这一部分在上文1.1 为iOS而设计（Designing for iOS）部分中已有详细描述，并寻求出如何在你的应用中融合与表达这种主题。当你这么做的时候，遵循本章中的指引可以帮助你为你的用户提供他们想要的体 验。

1.13.1 正确使用标准UI元素（Use Standard UI Elements Correctly）

尽可能使用UIKit提供的标准UI元素。多使用标准元素而非自定义元素，你与你的用户都将受益：

- 标准UI元素会根据iOS官方的更新而自动更新——而自定义元素不会。
- 标准UI元素对于你自定义外观和行为来说拥有优秀的扩展性。举个例子，iOS中所有的视图（Views）都是可自定义颜色的，它让应用配色变得很简单。想要了解更多如何给UI元素定义颜色，可以参考iOS 7 UI Transition Guide中Using Tint Color的相关章节。
- 用户更熟悉和习惯标准的元素，因为这对于他们来说没有学习成本，他们可以立刻明白这些元素的用途。

想要充分地利用标准UI元素的优点，有一些关键点需要特别注意：

严格遵循每个UI元素的设计规范。当你应用中的UI元素的外观与功能都是用户所熟悉的，他们可以很容易地根据先前的经验来使用他们，进而更好地使用你的应用。你可以从这些章节中找到各种UI元素的设计规范：Bars, Content Views, Controls, Temporary Views.

（译者注：上文提到的章节均处在iOS Human Interface Guidelines的第4章，翻译将在后续更新中放出，烦请各位耐心等候。若有需要，亦可先参考先前已翻译的iOS7 UI Elements章节：上，下。）

不要混用不同版本的iOS里的UI元素。你一定不希望让用户觉得你的UI元素来自于与当前设备版本不同的iOS系统。

大体来说，请避免创造自定义UI元素来表现标准交互行为。先 问问你自己为什么一定要创建一个与标准UI元素行为完全相同的自定义元素。如果你只是想改变标准UI元素的外观，可以考虑使用UIKit外观定制 API（UIKit appearance customization APIs），或者给元素填充别的颜色。如果你需要定义一个与标准控件稍有不同的行为，请确保你在改变了这个UI元素的属性和行为之后，这个元素仍然能完成 你所希望的操作。

不要用系统自带的按钮和图标表达其他含义。iOS提供了多种可用的按钮和图标。请确认你了解它们的准确表意；不要单纯凭借你看到这些图标样式的猜测和理解来解读和使用它们。（你可以在Toolbar and Navigation Bar Buttons和Tab Bar Icons中了解到这些按钮和图标的准确含义。）

如果你所需要的功能无法用系统提供的按钮和图标来表现，你也可以设计自定义按钮。自定义按钮的设计可以参考Bar Button Icons。

（译者注：上文提到的章节均处在iOS Human Interface Guidelines的第4章，翻译将在后续更新中放出，烦请各位耐心等候。若有需要，亦可先参考先前已翻译的iOS7 UI Elements章节：上，下。）

如果你的应用是沉浸式体验，那么创造全新的自定义UI是合理的。因为你在创造一个统一的体验环境，让用户在其中能够有所期待并探索如何控制应用。

1.13.2 弱化文件和文档处理（Downplay File and Document Handing）

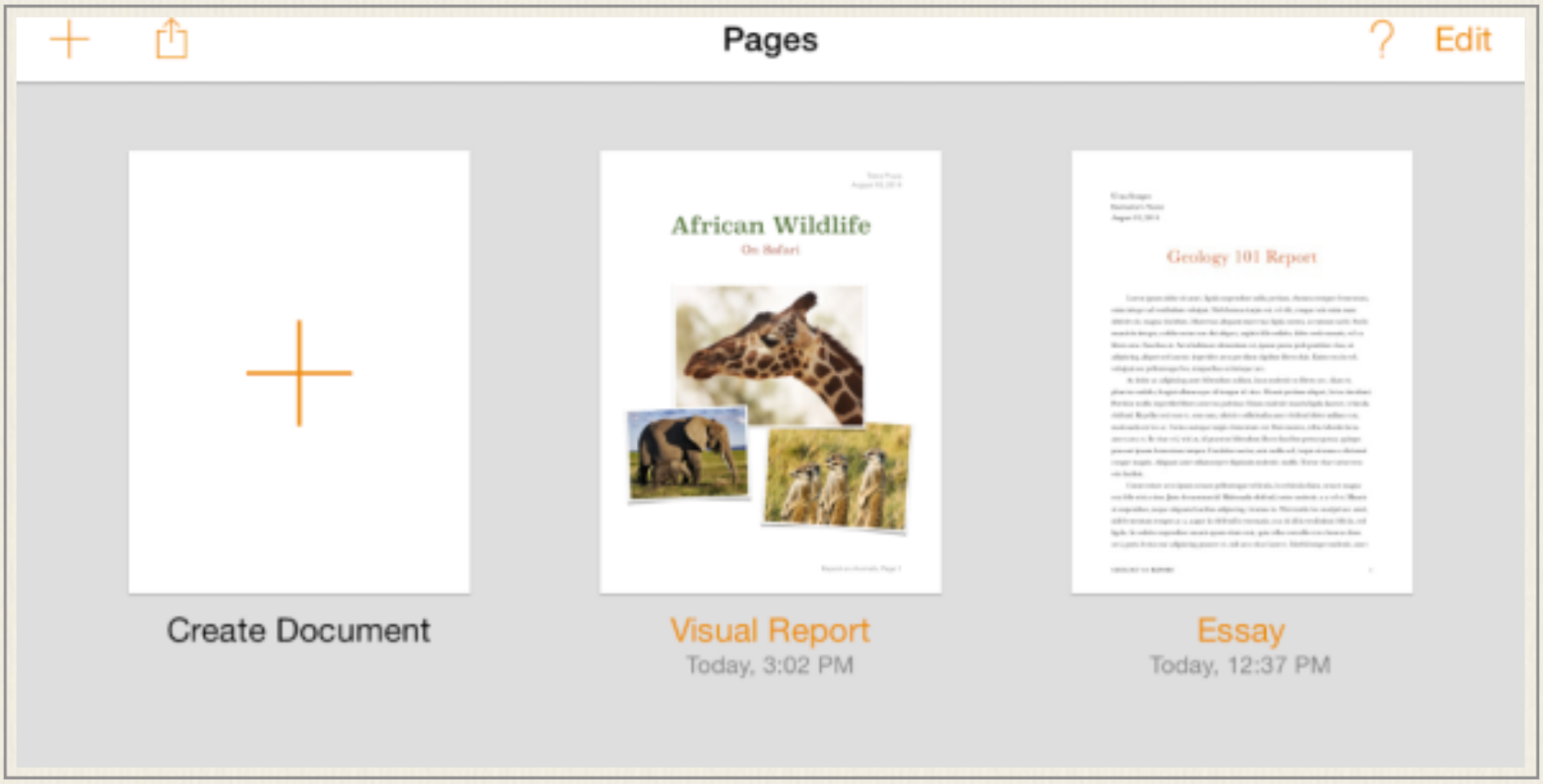
iOS应用可以帮助用户创建和处理文件，但这并不意味着用户需要过分考虑iOS设备的文件系统如何运作。

如果你的应用中支持用户创建和编辑文档，那么提供一个清晰的图形库视图（document library view）让用户能够方便地打开或者新建文档是一个好的做法。理想状况下，这样的图形库视图拥有以下特征：

- 高度图形化。用户通过屏幕上的缩略图就可以一目了然，快速找出自己想要的文件。

- 让用户用最少的动作完成自己的任务。比如说，用户可以快速地水平滚动文件列表，然后轻点一下自己想要的文件来打开它。
- 包含新建文档功能。一个图形库视图应该支持让用户点击一个新建文档的占位图便完成新建文档操作，而不是让用户通过访问别的地方来新建文档。

举个例子，Pages应用的图形库视图里面，既展示了用户已存在的文档，也加入了便捷的新建文档操作。



如果你的应用允许用户使用在其他应用中创建的文档，你可以通过模态文档选择视图控制器(modal document picker view controller)来帮助用户触达它们。这个控制器可以提取用户在iCloud中的文档，还可以通过文档提供者扩展（Document Provider extensions）来提取在其它应用中创建和储存的文件。想要了解更多文档提供者扩展的内容，可以参考Document Provider Extensions; 想要了解更多文档提取视图控制器，请参考Document Picker Programming Guide.

给用户足够的信心，让他们相信除非主动取消或者删除，他们的成果会被随时妥善保存。如果你的应用帮助用户创建于管理文档，不能要求用户每次都能及时保存。无论是打开另一个文档或切换应用的时候，iOS应用都应当承担起帮助用户保存输入内容的责任。

如果你的应用的主要功能不是创造内容，但又允许用户查看或编辑信息，这种情况下你需要询问用户是否要保存修改。在这种场景下，比较好的做法是提供 “编辑”按钮，点击后进入编辑状态，同时编辑按钮变成“保存”和“取消”按钮，这种变化可以提示用户当前处于编辑模式。“保存”可以保留修改内容，“取 消”则退出编辑模式。

1.13.3 必要时提供可配置选项（Be Configurable If Necessary）

某些应用需要用户手动安装或设置选项，但是大部分应用不需要如此。一个好的应用可以让大部分用户快速上手，并通过主界面给用户提供便捷的调整体验的方式。

当你的应用在默认状态下就能满足大部分用户的期望，用户对设置的需求就减少了。如果你需要储存用户的基本资料，可以优先向系统请求和拉取相关信息，而不是上来就让用户自己填写它。如果你一定要提供用户鲜少用到的设置项，请参考App Programming Guide for iOS中的The Setting Bundle部分来了解如何在代码中定义它们。

尽可能在主界面提供设置选项。如果用户在进行主线任务时有可能频繁改变设置，将设置项放在主UI中会很方便。如果用户只是偶尔才会用到设置项，那么可以将其放在独立的视图中。

如果应用内相关设置需要在系统设置中改变，帮助用户直接访问系统设置。尤其是，如果你要用一段文字来描述如何改变这个设置，比如“设置>隐私>定位服务”，倒不如直接放置一个按钮，点击后即可到达设置中的定位服务。想要了解如何实现，请参考 Settings Launch URL.

1.13.4 充分利用iOS技术（Take Advantage of iOS Technologies）

iOS提供了丰富的技术方式来支持用户完成他们所期望的各种任务和场景。这意味着在绝大多数情况下，将系统提供的技术整合到你的应用中，往往比自定义一种新的技术更为可靠。

某些iOS技术，比如多任务并行（Multitasking）与语音向导（VoiceOver）等等，是所有应用都应该包含的系统级特性。而另外一些技术是否整合到应用中，则取决于应用本身的功能性。比如处理门票和礼品卡的应用（Passbook）支持用户通过In-App Purchase完成购买，展示应用内置广告（iAd Rich Media Ads）则可以整合Game Center，同时支持iCloud。

英文原文访问地址：https://developer.apple.com/library/ios/documentation/userexperience/conceptual/mobilehig/index.html#//apple_ref/doc/uid/TP40006556-CH66-SW1

中文翻译PDF下载：http://url.cn/M4UiTq

原文链接：http://isux.tencent.com/ios8-human-interface-guidelines.html